Podstawy programowania
Materiały dydaktyczne do laboratorium

dr inż. K. M. Ocetkiewicz

27 października 2024

Zadania domowe

1 Parametry funkcji

W języku C parametry do funkcji przekazywane są wyłącznie przez wartość[1]. Oznacza to, że parametry zawierają kopię wartości argumentów[2]. Parametry funkcji można traktować tak samo jak zmienne lokalne funkcji – można je nawet modyfikować, jednak modyfikacja ta nie będzie widoczna na zewnątrz funkcji. Co jednak, gdy chcemy, aby funkcja zmodyfikowała zawartość przekazanych argumentów? Musimy skorzystać ze wskaźników. Wskaźniki także przekazywane są przez wartość, lecz różnica polega na tym, że mówimy funkcji nie co jest argumentem (a funkcja robi kopię tej wartości) lecz gdzie jest argument – parametr jest kopią przekazanego wskaźnika i wskazuje ten sam adres co argument.

Zadanie 1. Uruchom i przeanalizuj działanie poniższego kodu.

```
/* funkcja zamienia wartości przekazanych zmiennych */
#include<iostream>
using namespace std;

void ZamienNieDziala(int a, int b) {
  int t;
  t = a; a = b; b = t;
  };

void Zamien(int *a, int *b) {
```

[1] W języku C++ istnieje możliwość przekazania argumentu przez nazwę — należy skorzystać z referencji.

[2] Parametr to zmienna zawierająca wartość przekazaną do funkcji; argument to wartość przekazana do funkcji; np.

```
int F(int x, int y) { }
F(a+b, 4);
```

x i y są parametrami funkcji F, zaś a+b i 4 to argumenty w danym wywołaniu funkcji.

```
  int t;
  t = *a; *a = *b; *b = t;
  };

int main() {
  int a = 1, b = 2;
  cout << „a=" << a << „ b=" << b << endl;
  ZamienNieDziala(a, b);
  cout << „a=" << a << „ b=" << b << endl;
  Zamien(&a, &b);
  cout << „a=" << a << „ b=" << b << endl;
  return 0;
  };
```

Pozornie sprzeczny z przekazywaniem przez wartość jest sposób obsługi tablic.

Zadanie 2. Przeanalizuj i uruchom poniższy kod.

```
#include<iostream>

using namespace std;

void Wypisz(int tab[4], int rozmiar) {
  cout << „{„;
  for(int i = 0; i < rozmiar; i++) cout << „ „ << tab[i] << „,";
  cout << „}";
  };

void Ustaw(int tab[4], int pozycja, int wartosc) {
  tab[pozycja] = wartosc;
  };

int main() {
  int tab[4] = {1, 2, 3, 4};
  Wypisz(tab);
  cout << endl;
  Ustaw(tab, 2, 10);
  Wypisz(tab);
  cout << endl;
  return 0;
  };
```

Zadanie 3. Zmodyfikuj pierwszą linię funkcji Ustaw na (kolejno):

1. `void Ustaw(int tab[], int pozycja, int wartosc) {`
2. `void Ustaw(int *tab, int pozycja, int wartosc) {`

Zastanów się, jaki będzie rezultat w każdym przypadku i sprawdź, czy będzie tak w rzeczywistości. Dlaczego powinniśmy podawać rozmiar przekazywanej tablicy jako osobny parametr funkcji?

Parametry funkcji oraz jej zmienne lokalne są przechowywane na stosie. Jest to obszar pamięci wydzielony specjalnie do tego celu. Stos ma ograniczony rozmiar (domyślnie 1MB w systemach rodziny Windows i 10MB w systemach rodziny Linux). Wyczerpanie miejsca na stosie powoduje wystąpienie wyjątku.

Zadanie 4. Uruchom następujący kod.

```
#include<iostream>
using namespace std;

int main() {
  int tab[10485760];
  cout << "Hello world" << endl;
  return 0;
  };
```

Porównaj jego działanie z poniższym:

```
#include<iostream>
using namespace std;

int tab[10485760];
int main() {
  cout << "Hello world" << endl;
  return 0;
  };
```

Zmienne globalne nie są przechowywane na stosie, więc nie zużywają jego pamięci.

Czas życia zmiennych lokalnych jest ograniczony przez czas wykonania funkcji. Oznacza to, że zmienne te istnieją tylko wtedy, gdy uruchomiona jest funkcja je zawierająca.

Zadanie 5. Sprawdź rezultat wykonania poniższego kodu a następnie prześledź jego wykonanie debuggerem (jeżeli korzystasz z systemu Linux powiększ rozmiar każdej lokalnej tablicy dziesięciokrotnie):

```
#include<iostream>
using namespace std;

void F1() {
  char tab[400 * 1024];
  cout << "jestem w funkcji F1" << endl;
  };

void F2() {
  char tab[400 * 1024];
  cout << "jestem w funkcji F2" << endl;
  F1();
  };

void F3() {
  char tab[400 * 1024];
```

```
  cout << "jestem w funkcji F3" << endl;
  F2();
  };

int main() {
  F1();
  F2();
  F3();
  return 0;
  };
```

Każda z funkcji F1, F2, F3 potrzebuje 400KB pamięci stosu. Pierwsze wywołanie (F1()) zajmie 400KB na stosie. Gdy funkcja się zakończy, pamięć ta zostanie zwolniona i nadal będzie dostępne 1MB. Gdy wywołamy F2, funkcja ta zajmie 400KB po czym wywoła funkcję F1, która zużyje kolejne 400KB. Pozostawi to nieco ponad 200KB wolnej pamięci na stosie. Gdy zakończy się funkcja F1, 400KB zostanie zwolnione i gdy wykonanie wróci do funkcji F2, na stosie będzie ok. 600KB wolnej pamięci. Po zakończeniu wykonania F2 ponownie cały stos będzie dostępny. Ostatnie wywołanie w funkcji main (funkcja F3) będzie miało nieco inne konsekwencje. Gdy dojdziemy do momentu wywołania funkcji F1, cały czas będziemy „wewnątrz" F3 i F2, zatem na stosie musi być miejsce na zmienne lokalne wszystkich tych funkcji. O ile dla F2 i F3 miejsce jest (każda funkcja zużywa 400KB) to wywołanie F1 powoduje już wyczerpanie stosu i wystąpienie wyjątku.

2 Tablice wielowymiarowe jako argumenty

W przypadku parametrów funkcji będących tablicami wielowymiarowymi, należy pamiętać o kilku rzeczach. Pierwszą z nich jest różnica pomiędzy typem „wskaźnik na int" a „tablica intów o rozmiarze N". Jak widzieliśmy wcześniej, z punktu widzenia parametru funkcji oba typy są wymienne, a rozmiar samej tablicy nie ma znaczenia. Sytuacja komplikuje się jednak, gdy zwiększamy liczbę wymiarów naszej tablicy. Definiując zmienną:

```
int tablica[10][20];
```

tworzymy dwuwymiarową tablicę o wymiarach 10 na 20. Dokładniej, tworzymy tablicę o rozmiarze 10, której elementami są dwudziestoelementowe tablice liczb całkowitych. Jest to zmienna zupełnie innego typu niż:

```
int *tablica2[10];
```

która to zmienna jest dziesięcioelementową tablicą wskaźników. Zmienna tablica zajmuje 10*20*4 bajty, tablica2 zajmuje 10*4 bajtów (zakładając, że int i wskaźnik zajmują po 4 bajty). W przypadku parametrów do funkcji różnica pomiędzy wskaźnikiem a tablicą jest zatarta tylko w przypadku „najbardziej zewnętrznego" wymiaru (tego po lewej; jedynego, w przypadku funkcji jednowymiarowej). Wszystkie pozostałe muszą się zgadzać. Zatem poniższe wywołania będą poprawne:

```
void F1(int tab[][10]) { };
void F2(int (*tab)[10]) { };
void F3(int tab[99][10]) { };
...
```

```
int tablica[10][10];
F1(tablica); F2(tablica); F3(tablica);
```

natomiast te spowodują błąd kompilacji:

```
void F4(int tab[10][11]) { };
void F5(int *tab[10]) { };
void F6(int **tab) { };
...
int tablica[10][10];
F4(tablica); F5(tablica); F6(tablica);
```

Zwróć szczególną uwagę na różnicę pomiędzy typami:

```
int *tab1[10];
int (*tab2)[10];
```

tab1 jest dziesięcioelementową tablicą wskaźników na liczby całkowite; tab2 jest wskaźnikiem na dziesięcioelementową tablicę zmiennych całkowitych. tab1 to dziesięć wskaźników na inty; tab2 to jeden wskaźnik na tablicę dziesięciu intów.

Zadanie 6. Spójrz na poniższy kod:

```
void F1(int **tab) {};
void F2(int *tab[]) {};
void F3(int (*tab)[10]) {};
void F4(int (*tab)[20]) {};
void F5(int tab[10][10]) {};
void F6(int tab[10][20]) {};
void F7(int tab[20][10]) {};
void F8(int tab[][10]) {};

int main() {
  int tab[10][10];
  int (*ptab)[10];
  int *tabp[10];
  int **tabpp;
  F1(tab); F1(ptab); F1(tabp); F1(tabpp);
  F2(tab); F2(ptab); F2(tabp); F2(tabpp);
  F3(tab); F3(ptab); F3(tabp); F3(tabpp);
  F4(tab); F4(ptab); F4(tabp); F4(tabpp);
  F5(tab); F5(ptab); F5(tabp); F5(tabpp);
  F6(tab); F6(ptab); F6(tabp); F6(tabpp);
  F7(tab); F7(ptab); F7(tabp); F7(tabpp);
  F8(tab); F8(ptab); F8(tabp); F8(tabpp);
  return 0;
  };
```

Które z wywołań funkcji F1..F8 będą poprawne, a które spowodują błąd kompilacji?

3 Rekurencja

Rekurencja polega na wywołaniu funkcji przez samą siebie. Jest to bardzo przydatna technika i dostępna jest w każdym współczesnym języku programowania. Należy pamiętać, że każde kolejne wywołanie jest "samodzielne" – nie ma wpływu na poprzednie ani następne wywołania i każde z nich ma swój "zestaw" zmiennych lokalnych i parametrów.

Zadanie 7. Przeanalizuj i wykonaj poniższy kod:

```
#include<iostream>
using namespace std;

void Rekurencja(int p) {
  cout << "zaczyna␣sie␣wywolanie␣Rekurencja(" << p << ")" << endl;
  if(p == 0) {
    cout << "return␣z␣wywolania␣Rekurencja(" << p << ")" << endl;
    return;
    };
  Rekurencja(p - 1);
  cout << "konczy␣sie␣wywolanie␣Rekurencja(" << p << ")" << endl;
  };

int main() {
  Rekurencja(3);
  return 0;
  };
```

Zwróć uwagę, że wykonanie return kończy jedynie to wywołanie, w którym w danym momencie się znajdujemy. Zastanów się, dlaczego potrzebujemy fragmentu

```
if(p == 0) { return; };
```

Usuń ten fragment i zaobserwuj działanie programu (będziesz musiał(a) poczekać dłuższą chwilę). Zwróć także uwagę na ostrzeżenia zgłoszone podczas kompilacji.

Zadanie 8. Przeanalizuj poniższy kod. Policz, ile razy zostanie wywołana funkcja Fibonacci dla każdej wartości parametru n. Uruchom kod, i sprawdź, czy Twoje obliczenia zgadzają się z rzeczywistością. W przypadku rozbieżności, prześledź wykonanie kodu debuggerem.

```
#include<iostream>
using namespace std;

int Fibonacci(int n, int poziom) {
  int i, wynik;

  for(i = 0; i < poziom - 1; i++) cout << "|␣␣";
  if(poziom > 0) cout << "+->";
  cout << "obliczamy␣Fibonacci(" << n << ")" << endl;

  if(n <= 1) {
    for(i = 0; i < poziom - 1; i++) cout << "|␣␣";
```

```
    if(poziom > 0) cout << "+->";
    cout << "Fibonacci(" << n << ") = 1" << endl;
    return 1;
    };

  wynik = Fibonacci(n - 1, poziom + 1) + Fibonacci(n - 2, poziom + 1);

  for(i = 0; i < poziom - 1; i++) cout << "|  ";
  if(poziom > 0) cout << "+->";
  cout << "Fibonacci(" << n << ") = " << wynik << endl;

  return wynik;
  };

int main() {
  Fibonacci(5, 0);
  return 0;
  };
```

Każdy kolejne wywołanie zabiera nieco pamięci ze stosu, zatem po pewnym czasie miejsce się skończy i wystąpi wyjątek przepełnienia stosu. Należy pamiętać, że już samo wywołanie funkcji zużywa nieco pamięci stosu (zazwyczaj 8 bajtów w kodzie 32 bitowym, kilkadziesiąt bajtów w kodzie 64 bitowym). Czyli nawet funkcja, która nie ma zmiennych lokalnych, nie może wywoływać się rekurencyjnie w nieskończoność.

4 Podział programu na moduły

Programy komputerowe są bardzo długie. Powszechnie używaną jednostką jest kloc (kilo lines of code) czyli 1000 linii kodu. Jest to jednostka mająca zastosowanie nawet do niewielkich projektów. Np. projekt grupowy na naszej uczelni to zespół trzyosobowy, pół roku trwania, ok 20 dni implementacji przy średniej „wydajności" ok. 200 linii dziennie co daje 12kloc czyli 12 tysięcy linii. Trzymanie takiego kodu w jednym pliku spowoduje, że poruszanie się po nim stanie się praktycznie niemożliwe. Z tego powodu kod źródłowy dzielony jest na fragmenty nazywane modułami. Dodatkową zaletą podziału kodu programu na moduły jest skrócenie czasu kompilacji (w większości przypadków). W języku C moduł, formalnie nazywany jednostką kompilacji, jest synonimem jednego pliku źródłowego, po działaniu preprocesora, czyli włączając wszystkie pliki dołączone dyrektywą #include. Moduł to zbiór funkcji oraz (ewentualnie) zmiennych globalnych. Moduły mogą korzystać ze zmiennych globalnych oraz funkcji z innych modułów a także mogą udostępniać im swoje funkcje oraz zmienne. Funkcje zawarte w module powinny być logicznie powiązane ze sobą, zaś nazwa modułu powinna opisywać jego zawartość. Np. moduły o nazwie ekran, napisy, wejscie_wyjscie są dobre – wiemy, czego się spodziewać wewnątrz. Należy unikać modułów o nazwach typu: funkcje, rozne, modul1, modul2. Wszystko, co trzeba zrobić, aby utworzyć nowy moduł, to dodać do projektu nowy plik (czy w inny sposób spowodować, aby ten nowo utworzony plik był także kompilowany). W praktyce jednak wykonujemy nieco więcej pracy tworząc moduł, dzięki czemu samo użycie tego modułu staje się mniej pracochłonne. Aby to zrozumieć, musimy najpierw spojrzeć na proces zamiany kodu źródłowego na pliku wykonywalny.

5 Kompilacja

Proces tworzenia pliku wykonywalnego ma dwa etapy. Pierwszym z nich jest kompilacja. Jest to proces zamieniający plik źródłowy (każdy plik w projekcie jest kompilowany osobno) na postać półskompilowaną (są to pliki o rozszerzeniach .obj lub .o, w zależności od kompilatora). Każdy z takich plików zawiera skompilowany kod wszystkich funkcji oraz informacje o wszystkich zmiennych globalnych. Dodatkowo posiada listy eksportową i importową. Lista eksportowa wymienia funkcje i zmienne globalne, które dostępne są dla innych modułów. Lista importowa zawiera funkcje i zmienne, z których dany moduł korzysta, a które do niego nie należą. Zatem, jeżeli moduł A zawiera funkcję FA, a jedna z funkcji w module B korzysta z funkcji FA, to FA musi być na liście eksportowej modułu A oraz pojawi się na liście importowej modułu B. W języku C domyślnie wszystkie funkcje i zmienne globalne są eksportowane. Aby zmienić to zachowanie, należy skorzystać ze słowa kluczowego static (zob. kod przykładowy poniżej). Zaimportowanie funkcji jest nieco trudniejsze. Aby móc skorzystać z danej funkcji, kompilator nie musi znać jej definicji (jej treści). Musi jednak znać jej deklarację, czyli nazwę, typy parametrów oraz typ wartości zwracanej. Deklaracja funkcji to właściwie pierwsza linia jej definicji:

```
int Suma(int a, int b); // deklaracja funkcji
int Suma(int a, int b) { return a + b; }; // definicja funkcji
```

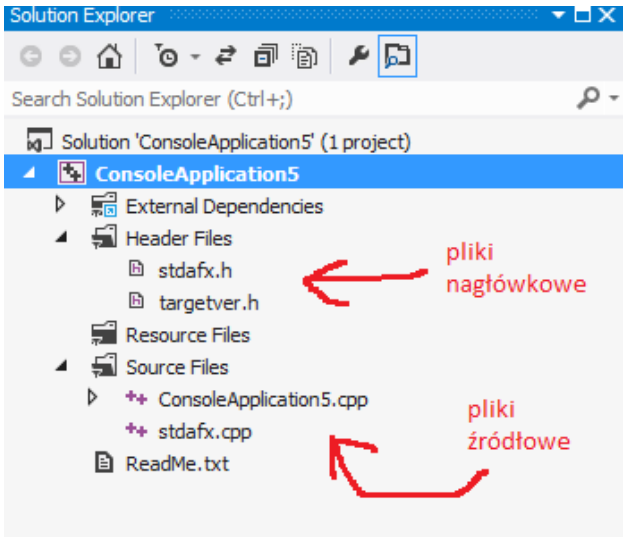
Definicja funkcji w całym kodzie programu może pojawić się tylko raz[3]. Deklaracja natomiast może pojawiać się wielokrotnie (nawet kilka razy w tym samym pliku). Definicja funkcji w danym module umieszcza w nim jej kod. Deklaracja funkcji informuje kompilator, że gdzieś (może w tym module, może w innym) pojawi się funkcja o danej nazwie i danych parametrach. W pierwszym etapie jest to informacja wystarczająca do prawidłowego skompilowania kodu. Aby zaimportować zmienną (zadeklarować ją zamiast definiowania) korzystamy ze słowa kluczowego extern:

```
extern int zmienna; // deklaracja zmiennej
int zmienna; // definicja zmiennej
```

Jeżeli zatem stworzymy moduł z pewną funkcją, którą wykorzystamy w innych modułach, to w tym module powinna znaleźć się definicja funkcji. Każdy moduł, który skorzysta z tej funkcji, musi znać jej deklarację. Aby nie powtarzać tej samej deklaracji w każdym z modułów, zazwyczaj umieszczamy deklaracje eksportowanych z modułu funkcji w osobnym pliku nagłówkowym (zazwyczaj z rozszerzeniem .h) w którym wypisujemy wszystkie deklaracje. W module, w którym korzystamy z funkcji z innego modułu dołączamy tylko plik nagłówkowy odpowiedniego modułu.

Przeznaczeniem plików nagłówkowych jest dołączenie ich w plikach źródłowych, zatem nie ma potrzeby kompilowania tych plików samodzielnie (zrobienie tego nie spowoduje jednak błędów). W różnych środowiskach zintegrowanych (np. Visual Studio czy Code::Blocks) pliki nagłówkowe umieszczamy nie pośród plików źródłowych, lecz w osobnej zakładce.

[3]Nie dotyczy to funkcji inline, które muszą być definiowane w każdym module, który z nich korzysta.



6 Konsolidacja

Gdy każdy moduł zostanie skompilowany, następuje konsolidacja („linkowanie”). Jest to połączenie wszystkich modułów, dodatkowych bibliotek oraz modułu uruchomieniowego (kodu, który wywołuje funkcję main) w jedną całość. Na tym etapie listy eksportowe zestawiane są z listami importowymi i konsolidator zamienia wywołania importowanych funkcji na wywołania konkretnych fragmentów kodu. Dopiero na tym etapie objawiają się niektóre błędy w naszym kodzie, np. gdy konsolidator nie znajdzie danej funkcji (ktoś ją importuje ale nikt nie eksportuje) czy więcej niż jeden moduł eksportuje funkcję lub zmienną o tej samej nazwie.

modulA.c

```
// korzystamy ze static: prywatnyLicznik i funkcja Suma nie będą
// widoczne poza modułem
static int prywatnyLicznik;
static int Suma(int a, int b) {
  return a + b;
  };

// definicja funkcji iloczyn
// z funkcji Iloczyn inne moduły będą mogły skorzystać
int Iloczyn(int a, int b) {
  return a * b;
  };
```

modulA.h

```
#ifndef modulA_h
#define modulA_h
// deklaracja funkcji Iloczyn
int Iloczyn(int a, int b);
#endif
```

```
modulB.c
#include"modul_A.h"
// funkcja Suma nie będzie widoczna poza modułem, więc nie
// będzie kolizji nazw
static int Suma(int a, int b) {
  return a * b;
  };

int main() {
  int a = Iloczyn(2, 3);
  return 0;
  };
```

Zadanie 9. Stwórz projekt zawierający powyższe pliki. Wykonaj następujące zmiany w powyższym kodzie (każda zmiana osobno) i zaobserwuj komunikaty o błędach:

1. Usuń linię #include“modulA.h” z modulB.c
2. Usuń deklarację funkcji Iloczyn z pliku modulA.h
3. Usuń słowa kluczowe static z definicji funkcji Suma w obu modułach.
4. Usuń definicję funkcji Iloczyn z pliku modulA.c (pozostaw deklarację w pliku nagłówkowym).

7 Funkcje a styl programistyczny

Funkcje są wykorzystywane w każdym programie (poza tymi wyjątkowo mało skomplikowanymi), dlatego też należy poświęcić czas na dopracowanie sposobu, w jaki się one prezentują.

Nazwa funkcji zazwyczaj jest czasownikiem (w odróżnieniu od zmiennych, których nazwy najczęściej są rzeczownikami), często w trybie rozkazującym. Wynika to z faktu, że tworząc funkcję chcemy zaimplementować pewną nową komendę czy polecenie, które będziemy wydawać komputerowi. Zatem powinniśmy nazywać funkcje np. Wypisz, WypiszNaEkran, CzytajZPliku czy, korzystając z języka angielskiego (co powinno być preferowane, dzięki temu unikniemy niejasności związanych z brakiem polskich liter) Print, PrintToScreen, ReadFile itd. Z drugiej strony powinien być to jeden czasownik. Czyli zamiast funkcji ObliczIWypisz czy NarysujWczytywane powinniśmy stworzyć osobno funkcje Oblicz, Wypisz, Narysuj i Wczytaj. Po pierwsze, takie funkcje będą prostsze (a tworząc prostą funkcję trudniej się pomylić), po drugie nie będziemy musieli duplikować kodu, gdy wystąpi kolejność narysowania wyniku obliczeń – wywołamy Oblicz i Narysuj, zamiast pisać funkcję ObliczINarysuj łącząc fragment z ObliczIWypisz z fragmentem funkcji NarysujWczytywane. Jeżeli jednak koniecznie potrzebujemy funkcję ObliczIWypisz, powinna to być funkcja wykorzystująca funkcje Oblicz i Wypisz:

```
void ObliczIWypisz() {
  ...
  wynik = Oblicz(...);
  Wypisz(wynik);
  };
```

zamiast:

```
int Oblicz() { /* obliczanie wyniku */ };
void Wypisz(int wynik) { /* wypisywanie wyniku */ };
int ObliczIWypisz() {
  /* obliczanie wyniku - to samo co w Oblicz */
  /* wypisywanie wyniku - to samo co w Wypisz */
  };
```

Nazwa funkcji powinna określać to, co ona robi. Porównaj poniższe fragmenty kodu, i zastanów się, co one robią:

```
// fragment 1
if(Sprawdzanie(x)) x = Przypadek1(x);
else x = Przypadek2(x);
. . .
// fragment 2
if(JestRabat(x)) x = ObliczRabat(x);
else x = DoliczPodatek(x);
```

Jeżeli chodzi o sam zapis nazw, to najpopularniejsze są trzy notacje:
`nazwaFunkcji` – tzw. camelCase – każdy wyraz nazwy, poza pierwszym, rozpoczynamy wielką literą, brak odstępów;
`NazwaFunkcji` – tzw. PascalCase – każdy wyraz rozpoczynamy wielką literą, brak odstępów;
`nazwa_funkcji` – standardowa C / GNU – oddzielamy wyrazy podkreślnikiem, nie stosujemy wielkich liter.

To, z której z powyższych notacji korzystamy jest mniej istotnie niż sam fakt konsekwentnego ich stosowania. Pamiętajmy, że jeżeli wyraźnie nie zaznaczymy przerw pomiędzy wyrazami, osoba analizująca kod może mylnie odczytać nazwę co bardzo utrudni jej zrozumienie kodu.

Funkcje nie powinny być długie. Dobrą praktyką jest aby długość funkcji nie przekraczała dwóch standardowych ekranów (czyli 48–50 linii). Oczywiście nie jest to absolutny nakaz, jednak jeżeli zauważymy, że funkcja przekracza ten rozmiar, należy się zastanowić nad podzieleniem jej na mniejsze fragmenty czy wydzieleniem niektórych elementów z wnętrza jako osobnych funkcji.

Pisząc deklarację funkcji należy podać nazwy jej parametrów. Ponadto nazwy te powinny być znaczące. Kompilator tego nie wymaga, ale jest to bardzo wygodne dla osoby korzystającej z kodu. Gdy szukamy w pliku nagłówkowym sygnatury funkcji (by dowiedzieć się, jak ją wywołać) i zobaczymy deklarację:

```
void Kopiuj(int *, int *, int);
```

czy nawet

```
void Kopiuj(int *a, int *b, int c);
```

będziemy wiedzieli znacznie mniej niż widząc:

```
void Kopiuj(int *dokad, int *skad, int ile);
```

a najlepiej:

```
Kopiuj(int const *dokad, const int const *skad, const int ile);
```

Rozważ ostatnią wersję – jak wskazówkę przypomnij sobie, że zmienne można czytać "od prawej do lewej" aby zapamiętać działanie modyfikatorów. Przykładowo:

```
int* p1;
const int* p2;
int* const p3;
const int* const p4;
```

$p1$ jest zwykłym wskaźnikiem na liczbę. $p2$ jest wskaźnikiem na stałą liczbę, tj. sam wskaźnik $p2$ może się zmieniać, lecz pamięć wskazywana nie. $p3$ jest stałym wskaźnikiem na liczbę, tj. sam wskaźnik $p3$ nie może się zmieniać, lecz pamięć wskazywana tak. $p4$ zaś jest stałym wskaźnikiem na stałą liczbę. Ostatni wariant może być najbardziej bezpieczny, gdyż zabezpiecza programistę przed przypadkową zmianą pamięci źródłowej lub też adresu tej pamięci. Również, przypadkowa zmiana ilości elementów kopiowanych lub też celu kopiowania jest cięższa.

W żadnym wypadku nie należy korzystać ze zmiennych globalnych do przekazywania parametrów do funkcji ani nie używać zmiennych globalnych jako zmiennych pomocniczych w funkcji. Znacznie obniża to czytelność kodu oraz utrudnia ponowne wykorzystanie funkcji (nie wystarczy przenieść funkcji ze starego projektu do nowego, trzeba także przenieść powiązane zmienne globalne i kod, który nadaje im początkowe wartości, a także upewnić się czy nie ma konfliktu nazw, co w przypadku zmiennych pomocniczych może się często zdarzyć). Porównaj kody:

```
// fragment 1
wiersz++;
zrodlo = source1;
cel = target1;
ile_razy = BUFFER_SIZE;
Copy();
cel = target2;
Copy();
...

// fragment 2
Copy(cel, zrodlo, BUFFER_SIZE);
Copy(cel2, zrodlo, BUFFER_SIZE);
```

W pierwszym fragmencie, dopóki nie przeanalizujemy funkcji Wypisz, nie mamy pewności, czy przypisanie do zmiennej *wiersz* ma związek z kopiowaniem. Nie jesteśmy rownież pewni intencji programisty. Czy funkcja copy zmienia *ile_razy*, albo zrodlo – czy powinna to robić – a może jest tu błąd? W drugim przypadku intencja programisty jest o wiele bardzie czytelna.

Dobrym zwyczajem jest opisanie funkcji w komentarzu umieszczonym bezpośrednio przed funkcją. Dzięki niemu od razu wiemy czego się spodziewać. Dodatkowo, istnieją narzędzia (np. Doxygen[4]) które potrafią z takich komentarzy zbudować bardzo estetyczną dokumentację kodu, np.:

```
/** wypisanie na ekran, we wskazanym miejscu, znaku w podanym
    kolorze
```

[4]http://www.stack.nl/ dimitri/doxygen/

```
    \param x kolumna, w której zostanie wypisany znak
    \param y wiersz, w którym zostanie wypisany znak
    \param z znak do wypisania
    \param c kolor znaku
    \return znak który poprzednio znajdował się w miejscu
            (x, y) */
char WypiszZnak(int x, int y, char z, int c) {
  // treść funkcji
  };
```

Powyższy komentarz jest przykładem opisu funkcji dla narzędzia Doxygen, jednocześnie bardzo czytelnie opisuje dzianie funkcji.

Komentarz taki może wskazać nam potencjalne problemy — jeżeli opis działania jest za długi (więcej niż jedno-dwa zdania) lub lista parametrów zajmuje za dużo miejsca (zazwyczaj funkcje nie powinny mieć więcej niż 4–6 parametrów) sugeruje to, że powinniśmy przeprojektować funkcję czy podzielić ją na mniejsze fragmenty.

8 Ćwiczenia

Zadanie 10. Napisz funkcję rekurencyjną obliczającą sumę elementów tablicy w następujących wariantach:

1. w każdym kroku oddzielając pierwszy element tablicy
2. w każdym kroku oddzielając ostatni element tablicy
3. (*) w każdym kroku wyłączając środkowy element i dzieląc pozostałą tablicę na dwie prawie równe części

Zadanie 11. Napisz rekurencyjną funkcję znajdującą zadany element tablicy. Funkcja powinna zwracać indeks znalezionego elementu lub -1 gdy nie występuje on w tablicy.

Zadanie 12. Napisz funkcję rekurencyjną zamieniającą pierwsze wystąpienie podanej wartości na inną.

Zadanie 13. Napisz funkcję rekurencyjną odwracającą kolejność elementów w podanej tablicy.

Zadanie 14. Napisz funkcję sprawdzającą, czy dana tablica 4x4 zawiera kwadrat magiczny. Do sprawdzania równości wszystkich sum użyj dwóch pętli for. Wskazówka: zapisz w dodatkowych tablicach punkty początkowe i kierunki liczenia każdej sumy.
(zob. http://pl.wikipedia.org/wiki/Kwadrat_magiczny_%28matematyka%29)

Zadanie 15. Napisz program zamieniający linie narysowane gwiazdkami w dwuwymiarowej tablicy na linie rysowane znakami -, |, + i wyświetlający wynikowy rysunek na ekran. Np.

```
to:             zamieniamy na to:
***********     +---------+
*    *    *     |    |    |
* ** * ** *     | +- | -+ |
* *  *  * *     | |  |  | |
***********     |-+--+--+-|
*    *    *     |    |    |
*      *  *     |      |  |
* ***  *  *     | ---  |  |
*      *  *     |      |  |
*         *     |         |
***********     +---------+
```

Skorzystaj z szablonu:

```
void Zamien(char we[][12], char wy[][12]) {
  };

void Wypisz(char tab[][12]) {
  };

int main() {
  char we[11][12] = {
    "***********",
    "*␣␣␣␣*␣␣␣␣*",
    "*␣**␣*␣**␣*",
    "*␣*␣␣*␣␣*␣*",
    "***********",
    "*␣␣␣␣*␣␣␣␣*",
    "*␣␣␣␣␣␣*␣␣*",
    "*␣***␣␣*␣␣*",
    "*␣␣␣␣␣␣*␣␣*",
    "*␣␣␣␣␣␣␣␣␣*",
    "***********"
    };
  char wy[11][12];
  Zamien(we, wy);
  Wypisz(wy);
  return 0;
  };
```

Zadanie 16. Zmodyfikuj rozwiązanie poprzedniego zadania tak, aby funkcje Zamien i Wypisz znajdowały się w osobnym pliku źródłowym.

Zadanie 17. Zaimplementuj grę w życie Conway'a na planszy ograniczonej do wymiarów 70 (szerokość) na 23 (wysokość). Po każdym kroku wyświetlaj zawartość planszy na ekranie. Pamiętaj o poprawnej obsłudze brzegów planszy.
(zob. http://pl.wikipedia.org/wiki/Gra_w_%C5%BCycie)

Zadanie 18. Zmodyfikuj poprzednie zadanie tak, aby reguły można było modyfikować w łatwy sposób (np. korzystając z tablicy liczb sąsiadów, dla których komórka ożywa i tablicy liczb sąsiadów, dla których komórka pozostaje żywa).

Zadanie 19. Podziel powyższy program na moduły. Umieść funkcję main w jednym pliku, funkcje związane z regułami gry w drugim, funkcje wyświetlające na ekranie w trzecim pliku.