WYDZIAŁ ELEKTROTECHNIKI I AUTOMATYKI

Wybrane mechanizmy programowania w językach C/C++

Autorzy:
Robert Smyk - redaktor
Maciej Czyżak
Artur Opaliński

Publikacja jest dystrybuowana bezpłatnie.

Materiał został przygotowany w związku z realizacją projektu pt. „Zamawianie kształcenia na kierunkach technicznych, matematycznych i przyrodniczych – pilotaż" współfinansowanego ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego Nr umowy: 46/DSW/4.1.2/2008 – zadanie 018240 w okresie od 21.08.2008 – 15.03.2012

Spis treści

Część I Podstawy języka C – Maciej Czyżak

1 Charakterystyka języka C i przykłady prostych programów.

Język C powstał na początku lat 70-tych. Definicja języka C została przedstawiona przez Dennisa M. Ritchie w roku 1972. Jako pierwowzór posłużył beztypowy język B (Ken Thompson (1970)) będący adaptacją języka BCPL (Basic Combined Programming Language, Martin Richards(1967)) dla PDP-7. W roku 1974 opis języka C zaprezentowano w opracowaniu D.M. Ritchie, B.W. Kernighan, The C programming language, Prentice-Hall. Wersja języka C przedstawiona tam zwana jest standardem K&R.

W roku 1988 pojawił się nowy standard języka C- język ANSI C. Opis tego standardu zawarty został w dokumencie American National Standard for Information Systems - Programming Language C, X3.159-1989. W roku 1990 powstał też standard ISO 9899:1990. Kolejna wersja standardu pojawiła się roku 1999, jest ona określana jako C99.

Powodem stworzenia języka C była potrzeba zastąpienia, w możliwie dużym stopniu, języka asemblera językiem wyższego poziomu przy tworzeniu systemów operacyjnych (początkowo dotyczyło to systemu UNIX) i oprogramowania narzędziowego takiego jak kompilatory, programy łączące (ang. linker) jak również edytory i procesory tekstu. Istniejące w tamtym okresie języki takie jak FORTRAN, ALGOL 60 czy też PL/I nie nadawały się do tych celów. Elementy języka C weszły jako podzbiór do szeregu nowszych języków takich jak C++, Java, C# i PHP. Język C stosowany jest szeroko do programowania procesorów sygnałowych i sterujących. Daje om programy zwykle kilkakrotnie szybsze niż w nowszych językach, z tego powodu jest stosowany tam, gdzie istnieją duże wymagania czasowe.

Wstęp do programowania

Komputer, którego podstawowe koncepcje opracowano w latach 30-tych i 40-tych XX w., jest urządzeniem istotnie różniącym się od maszyn znanych wcześniej, które przeznaczone były do wykonywania jednej lub kilku ustalonych operacji. Komputer natomiast jest maszyną elastyczną, która może być stosowana np. do sterowania silnikiem, realizacji obliczeń matematycznych, analizy obrazów, odtwarzania filmów i dźwięku, sterowania smartfonem czy też w grach komputerowych. Elastyczność taka wynika stąd, że każdy komputer posiada pewną liczbę elementarnych poleceń zwanych rozkazami (instrukcjami). Zbiór rozkazów stanowi tzw. listę rozkazów. Rozkazy z tej listy mogą być zestawiane w odpowiednie sekwencje umożliwiające realizację odpowiednich zadań. Sekwencja rozkazów stanowi program. Proces tworzenia programu zwany jest programowaniem. W początkowej fazie rozwoju komputerów rozkazy nie były jeszcze dostępne, stąd komputery programowano łącząc kablami odpowiednie obwody komputera. W kolejnej fazie rozwoju, gdy dostępne już były rozkazy w postaci ciągów zerojedynkowych (np. 40-bitowych), programy tworzono jako odpowiednie sekwencje takich ciągów, programowanie takie zwane jest programowaniem w języku wewnętrznym. Kolejny etap stanowiło programowanie w języku asemblera. W podejściu tym fragmenty rozkazów w języku wewnętrznym zastępowano skrótami literowymi tzw. mnemonikami. Programowanie w języku asemblera jest bardzo czasochłonne, co

stało się powodem stworzenia języków algorytmicznych (strukturalnych), gdzie program jest zapisywany w sposób bardziej zbliżony do języka naturalnego. Rosnąca komplikacja i rozmiary oprogramowania spowodowały powstanie programowania obiektowego, gdzie istnieje możliwość operowania złożonymi strukturami zwanymi klasami i obiektami i korzystania z bogatych bibliotek klas zawierających tysiące elementów.

Podstawowe elementy języka to:

Typy:

- typy podstawowe : typy znakowe, całkowite i rzeczywiste
- typy pochodne : wskaźnikowe, tablice, struktury, unie i inne

Wyrażenia : budowane z operatorów i ich argumentów

Wskaźniki : operacje na adresach niezależnie od maszyny

Konstrukcje sterujące:

- grupowanie instrukcji (instrukcja złożona)
- podejmowanie decyzji (if, if-else, switch)
- realizacja powtarzania ze sprawdzaniem warunku na
- początku (for, while), na końcu (do-while)
- przerwanie (break) i kontynuacja pętli (continue)

Ogólna budowa programu

- program składa z pewnych jednostek programowych (fragmentów kodu) zwanych funkcjami z dobrze określonymi mechanizmami komunikacji z otoczeniem:
- program w C posiada tzw. płaską strukturę tzn. funkcje są niezależne i nie można definiować jednych funkcji w drugich.

Funkcje:

- mogą zwracać wartości typów podstawowych, struktury, unie i wskaźniki
- zmienne w funkcjach mogą być zmiennymi lokalnymi (automatyczne i statyczne) lub zewnętrznymi
- mogą być wywoływane rekurencyjnie
- mogą być rozmieszczone w jednym lub większej liczbie plików

Przykład 1. Prosty program w języku C - wersja 1.

```
1. #include <stdio.h>
2. int main(int argc, char* argv[])
3. {
```

```
4. printf("Pierwszy program w jezyku C");
5. return 0;
6. }
```

Komentarz do przykładu 1.

Powyższy program drukuje napis ”Pierwszy program w jezyku C” bez litery 'ę'. Przytoczony przykład pokazuje ogólną budowę najprostszego programu w języku C.

W języku C program składa się z segmentów, zwanych funkcjami, zawierających kod, który może być wykonywany. Kompletny program musi zawierać funkcję o nazwie main(), od której rozpoczyna się wykonanie programu. Funkcja ta może być umieszczona w dowolnym miejscu pliku zawierającego tekst programu(pliku źródłowego). Ze względu na to, iż konstrukcje języka nie obejmują wszystkich elementów, które są konieczne, aby program mógł być realizować niektóre działania (np. wydruk na ekranie), konieczne jest dołączenie do programu plików zawierających pewne symbole i nagłówki funkcji zdefiniowanych w innych miejscach. Dokonuje się tego za pomocą dyrektywy #include <nazwa_pliku>. Dyrektywa #include <stdio.h> w linii 1 oznacza polecenie włączenia do programu pliku stdio.h. Nazwa pliku jest skrótem od ang. Standard Input Output, co oznacza standardowe wejście-wyjście. Nawiasy < > oznaczają, że jest to tzw. plik systemowy, co oznacza, że jego położenie w systemie plików jest kompilatorowi znane (jeśli chciałoby dołączyć własny plik należałoby zastosować konstrukcję #include ”mojPlik.h”). Plik stdio.h zawiera nagłówki standardowych funkcji wejścia/wyjścia. Jedną z nich jest funkcja printf służąca do wypisywania wartości różnych typów na ekranie. Każda instrukcja w języku C musi być zakończona średnikiem ';'. Instrukcje składające się na kod funkcji umieszcza się w nawiasach klamrowych '{', '}'. Poszczególne linie programu mają następujące działanie:

Linia 1 #include <stdio.h>

Jest tzw. linią sterującą preprocesora i wymaga ona wykonania pewnego działania, zanim program zostanie przetłumaczony na kod maszynowy. Preprocesor to specjalny program przetwarzajacy tekst programu przed przekazaniem go kompilatorowi. Linia sterująca preprocesora rozpoczyna się zawsze znakiem #. Powoduje ona włączenie (wstawienie) zawartości wymienionego pliku (w tym przypadku stdio.h), dokładnie w miejscu, gdzie pojawia się linia #include. Plik stdio.h zapewnia właściwy interfejs do funkcji bibliotecznej printf.

Linia 2 `int main(int argc, char* argv[])`

Linia ta to nagłówek funkcji main(). Pierwszym elementem nagłówka jest słowo int, będące skrótem od słowa ang. integer oznaczającego “całkowity”. Rola konstrukcji z tym słowem wynika z działania funkcji main( ). Funkcja ta, gdy zakończy swoje działanie wysyła do systemu operacyjnego pewną wartość (zwykle 0) i słowo int jest określeniem typu wysyłanej wartości. Słowo main oznacza nazwę funkcji. Kolejnym elementem nagłówka jest

`(int argc, char* argv[])`

Konstrukcja ta zwana jest listą parametrów funkcji umieszczonych w nawiasach. Składa się ona z typów parametrów i ich nazw. Pierwszy parametr argc służy do przekazania liczby argumentów użytych przy wywołaniu (uruchomieniu programu) w trybie konsoli, a parametr argv do przekazania słów użytych w linii przy uruchomieniu programu. Jest to tablica, której elementami są tablice znakowe. W języku C dla tablicy znakowej stosuje się określenia łańcuch znaków i tekst.

Znaczenie tej konstrukcji będzie omówione w p.1.13 poświęconym tablicom znakowym.

Linia 3 {

Linia ta zawiera nawias klamrowy rozpoczynający treść funkcji, nawias kończący jest umieszczony w Linii 6.

Linia 4 `printf("Pierwszy program w jezyku C");`

W linii tej znajduje się instrukcja wywołania (uruchomienia) funkcji printf mającej za zadanie wydrukowanie napisu na ekranie. Wywołanie takie składa się z nazwy funkcji oraz argumentu (lub kilku argumentów) zawartych w nawiasach ( ). Tutaj argumentem jest tablica znakowa "Pierwszy program w jezyku C", ograniczona cudzysłowami. Średnik umieszczony na końcu stanowi zakończenie instrukcji.

Linia 5 `return 0;`

Słowo ang. return oznacza *powróć* i *zwróć*. Stanowi ono część instrukcji przekazującej sterowanie do programu uruchamiającego dany program, tutaj systemu operacyjnego. Dodatkowo instrukcja return powoduje zwrócenie (wysłanie) wartości 0 do systemu operacyjnego.

Wadą tego programu jest to, że wynik wyświetlany w oknie konsoli, znika. Poniżej zostaną pokazane sposoby zatrzymania wyświetlanego rezultatu działania programu.

Przykład 2. *Prosty program w języku C - wersja 2 (zatrzymywanie wykonania programu).*

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main(int argc, char* argv[])
4. {
5. printf("Pierwszy program w jezyku C");
6. getch();/* lub system("pause") lub getchar()*/
7. return 0;
8. }
```

Komentarz do przykładu 2 (w dalszej części będą komentowane tylko linie zawierające nowe elementy)

Linia 2 `#include <conio.h>`

W linii tej umieszczono polecenie włączenia pliku conio.h, zawierającego funkcję getch() , zasadniczo wczytującą pojedynczy znak, jednak tutaj wczytywany znak nie jest wykorzystywany, używana jest właściwość tej funkcji polegającą na zatrzymaniu biegu programu i oczekiwaniu na wprowadzenie jakiegoś znaku przez użytkownika.

Linia 6 `getch();/* lub system("pause") lub getchar()*/`

Linia ta zawiera polecenie uruchomienia funkcji getch(), co daje efekt zatrzymania wykonywania programu, ponadto w komentarzu w tej linii zamieszczono inne funkcje umożliwiające zatrzymanie programu. Komentarz jest tekstem umieszczanym między parami znaków /* i */. W wersji C99 wprowadzono też komentarz jednoliniowy, rozpoczynający się się parą znaków // i rozciągający się do końca linii. Komentarze nie wpływają na kod programu.

Przykład 3. *Prosty program w języku C - wersja 3.*

```
1. //program ten jest uzupełniony o wybrane elementy
2. #include <stdio.h>
3. #include <conio.h>
4. int main(int argc, char* argv[])
5. {
6. system("cls");/* lub clrscr(), w DevC++ konieczny plik naglowkowy conio2.h*/
7. printf("Pierwszy program w języku C");
8. printf("\n napisany przez J. Kowalskiego");
9. getch();
   return 0;
10. }
```

Komentarz do przykładu 3

Linia 2 `system("cls");`

W linii tej znajduje się wywołanie funkcji `system` z argumentem "cls". Spowoduje ona przekazanie do systemu operacyjnego polecenia wyczyszczenia ekranu. Podano też drugą funkcję umożliwiającą czyszczenie ekranu.

Przykład 4. *Program z użyciem zmiennych typu* `int`.

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main(int argc, char* argv[])
4. {
5. int a;/* lub int a,b,c;*/
6. int b;
7. int c;
8. a=1;
9. b=2;
10. c=a+b;
11. printf(" Suma a+b =%d",c);
12. getch();
13. return 0;
14. }
```

Komentarz do przykładu 4

Linie 5, 6.7 `int a; int b; int c;`

W liniach tych zamieszczono definicje trzech zmiennych `a`, `b`, `c`. Definicja zmiennej oznacza zarezerwowanie komórki pamięci komputera, w której można przechowywać pewną wartość podczas działania programu. Każda z tych definicji składa się z nazwy typu (tutaj typ `int`) danej

zmiennej i jej nazwy. Zmienna typu int może przechowywać wartości całkowite. Dopuszczalny zakres tych wartości jest podany w pkt. 1.2. w tabeli 1 (str. 12). Nazwy zmiennych (identyfikatory) powinny się zaczynać od litery lub znaku podkreślenia i mogą zawierać litery, znaki podkreślenia i cyfry. Nazwy zmiennych nie mogą zawierać spacji i innych znaków. Szczegóły tworzenia nazw podano w pkt.1.6 (str. 23). W ramach danego bloku utworzonego przez nawiasy klamrowe { } nazwy zmiennych nie mogą się powtarzać.

Linia 8 `a=1;`

Instrukcja w tej linii jest instrukcją przypisania (podstawienia). Użyty symbol '=' jest operatorem przypisania. Instrukcja ta oznacza zapis wartości 1 do zmiennej a (do komórki pamięci oznaczonej jako a). Liczba 1 jest stałą całkowitą.

Linia 9 `b=2;`

W linii tej zmiennej b przypisywana jest wartość 2.

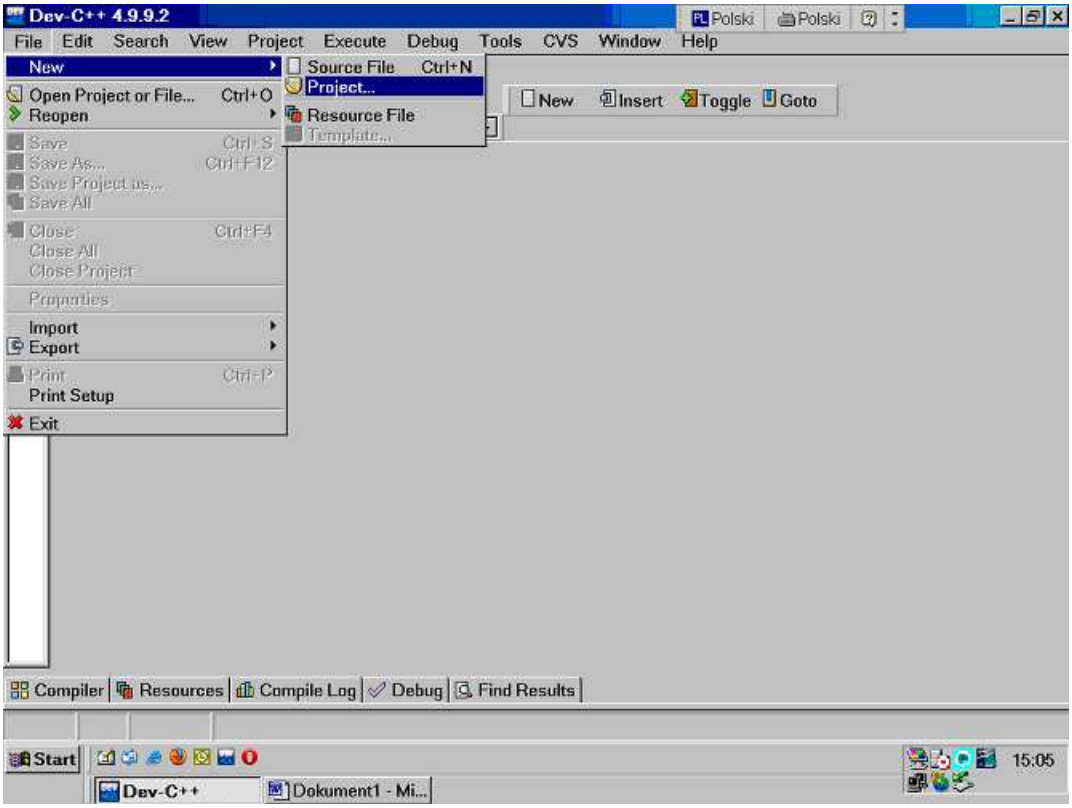
Linia 10 `c=a+b;`

Linia zawiera po prawej stronie wyrażenie złożone z dwóch operandów, którymi są zmienne a i b oraz operator dodawania całkowitego '+'. Podczas wykonywania tej instrukcji najpierw pobierane są z pamięci wartości zmiennych a i b (umieszczone w komórkach pamięci oznaczonych jako a i b) , następnie obliczana jest suma, następnie wartość tej sumy jest podstawiana do komórki c. Nie w każdym przypadku uzyska się wartość poprawną, wynika to z faktu, że wartość sumy musi się mieścić w zakresie zmiennych typu int.

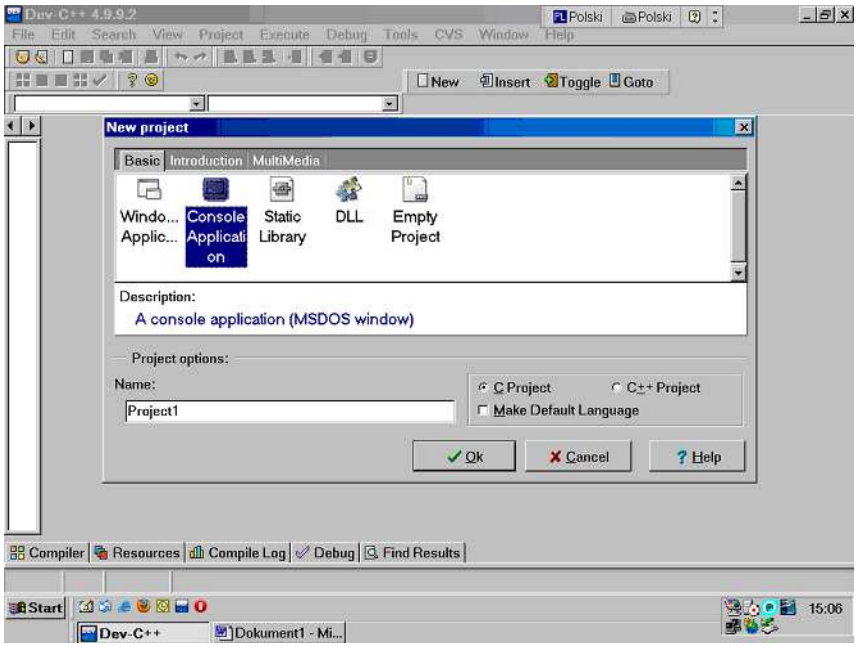
Linia 11 `printf(" Suma a+b =%d",c);`

W tej linii funkcja printf ma dwa argumenty, pierwszy to " Suma a+b =%d" i drugi zmienna c. Zadaniem funkcji printf jest wyprowadzenie na ekran napisu Suma a+b = i wartości zmiennej c w postaci dziesiętnej. Para symboli %d reprezentuje tzw. deskryptor formatu. Format oznacza zewnętrzną reprezentację danej wartości (zwykle na ekranie), zaś deskryptor opis tego formatu. Deskryptor %d mówi tutaj, że wartość zawartą w zmiennej c należy wyprowadzić na ekran jako liczbę całkowitą dziesiętną.

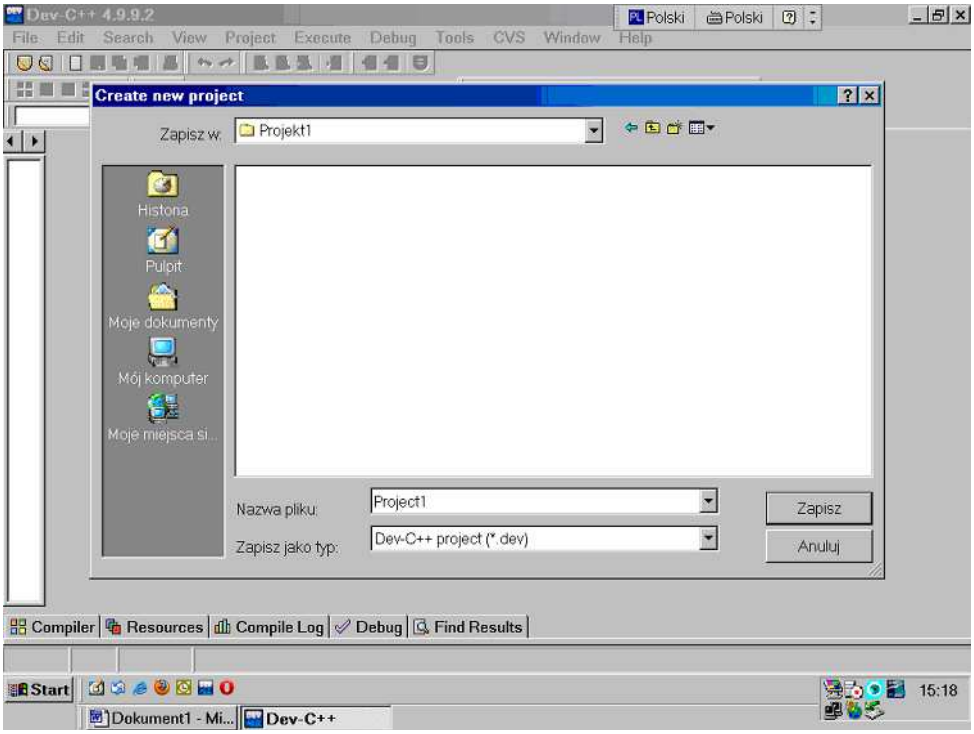
Program w języku C przed wykonaniem musi być poddany kompilacji i następnie konsolidacji. Konsolidacja zwana też łączeniem (ang, linking), jest proces to polegający na dołączeniu do kodu programu elementów, które są konieczne dla stworzenia programu wykonywalnego (w systemie Windows .exe). Na każdym z tych etapów mogą się pojawić błędy, które muszą być usunięte. Kompilatory i konsolidatory dla języka C są dostępne praktycznie w każdym systemie operacyjnym. Poniżej zaprezentowane będą podstawy tworzenia programu przy użyciu środowiska DevC+ 4.9.2.2 na licencji GNU General Public Licence. Środowisko to należy pobrać nieodpłatnie ze strony **http://www.bloodshed.net/dev/devcpp.html** i następnie zainstalować. Po uruchomieniu środowiska otrzymujemy ekran przedstawiony poniżej.



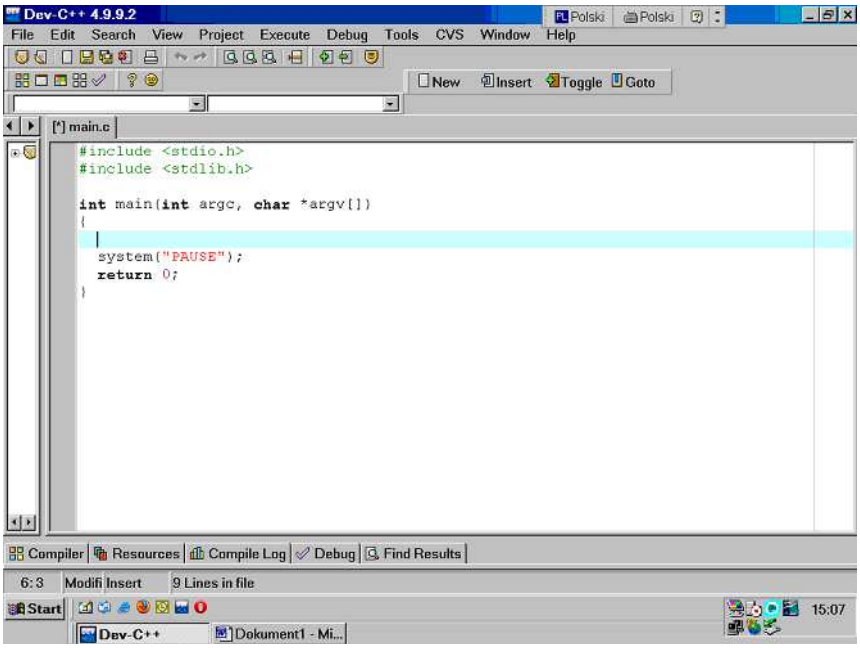
Należy wybrać opcję **File**, podopcję **New**, a następnie podopcję **Project**. Po jej wybraniu uzyskujemy kolejny ekran. Zakładając, że naszym celem jest stworzenie aplikacji pracującej w trybie tekstowym, należy wybrać opcję **Console Application**.



Po wybraniu tej opcji należy zapisać projekt, najlepiej w nowo utworzonym katalogu.



Po zapisaniu projektu następuje automatyczne przejście do trybu, w którym możemy tworzyć tekst programu, kompilować i uruchamiać.



Aby program skompilować należy wybrać opcję **Execute** i następnie **Compile**, aby skompilować program i **Run** by program uruchomić. Można też połączyć te działania wybierając opcję **Compile&Run**.

2 Zmienna, typy całkowite i typy rzeczywiste

Zmienna w języku C, z punktu widzenia programisty, oznacza pewien **obszar pamięci komputera**, który może być używany przez programistę w określony sposób. Np. dla zmiennej a w przykładzie 4, rozmiar tego obszaru określa typ int. Typ ten oznacza, że w zmiennych tego typu można przechowywać wartości całkowite z zakresu $[\ -2^{31}\ ,\ 2^{31}-1\ ]$ (w komputerach 32-bitowych typ int zajmuje 4 bajty). Liczby całkowite są przechowywane w kodzie z uzupełnieniem do 2 (U2).

Typ danej wartości lub zmiennej określa jej budowę (liczbę bajtów, wewnętrzny format), a także możliwości użycia w programie. Do typów podstawowych należą typy całkowite, typy rzeczywiste i typy znakowe.

Typy całkowite są opisywane przez nazwę typu, rozmiar w bitach (lub bajtach) oraz zakres liczbowy, który wynika z rozmiaru i tego, czy typ jest typem ze znakiem.

Tabela 1. Typy całkowite

Typ	***Rozmiar(bity)***	***Zakres liczbowy***
unsigned char	8	*0 do 255*
char	8	*-128 do 127*
short int	16	*-32768 do 32767*
unsigned short int	16	*0 do 65535*
int	32	*-2147483648 do 2147483647*
unsigned int	32	*0 do 4294967295*
long int	32	*zakres int*
unsigned long int	32	*zakres unsigned int*
long long int	64	-2^{63} do $2^{63}-1$
unsigned long long int	64	0 do $2^{64}-1$

unsigned oznacza typ bez znaku (kod NKB), pozostałe są przechowywane w kodzie U2**.**

Typy rzeczywiste są opisywane przez nazwę typu, rozmiar w bitach (lub bajtach) oraz zakres liczbowy i maksymalną precyzję określającą liczbę miejsc dziesiętnych (cyfr), jakie można uzyskać na wydruku.

Tabela 2. Typy rzeczywiste

Typ	***Rozmiar (bity)***	***Zakres liczbowy***	***Liczba miejsc znaczących***
float	32	$3.4*10^{-38}..3.4*10^{38}$	6 - 7
double	64	$0.7*10^{-38}..3.4*10^{38}$	15 - 16
long double	80	$3.4*10^{-4932}..3.4*10^{4932}$	19 - 20

3 Operatory arytmetyki liczb całkowitych i arytmetyki liczb rzeczywistych

Operatory to znaki lub kombinacje znaków symbolizujące pewne operacje w programie np. **=,**

+, -. Operatory służą wraz i innymi elementami do tworzenia tzw. **wyrażeń**. Wyrażenie jest pewną kombinacją operatorów i innych elementów takich jak zmienne i stałe.

Przykładami wyrażeń mogą być:

-a -jest to wyrażenie składające się z operatora minus (-) i argumentu (zmiennej **a**)

-b+c -wyrażenie to składa się z operatora plus(+) i dwóch argumentów (zmienne **a** i **b**)

Operator minus(-) w pierwszym wyrażeniu jest operatorem jednoargumentowym, a operator plus(+) w drugim wyrażeniu jest operatorem dwuargumentowym.

Ogólnie w języku C operator jednoargumentowy może być użyty w sposób następujący:

op argument lub argument op,

a operator dwuargumentowy

argument1 op argument2

W powyższych konstrukcjach **op** oznacza operand.

Operatory arytmetyki całkowitej

Do operatorów arytmetyki całkowitej należą:

-operatory jednoargumentowe ‘+’ i ‘ –’

-operator dodawania ‘+’

-operator odejmowania ‘-’

-operator mnożenia ‘*’

-operator dzielenia całkowitego ‘/’

-operator obliczania reszty z dzielenia całkowitego '%'

Operatory te są stosowane, gdy argument lub argumenty operatora są typu całkowitego.

Przykład 5. Program ilustrujący użycie operatorów arytmetyki całkowitej dla liczb ze znakiem.

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main(int argc, char* argv[])
4. {
5. int a,b,c;
6. printf("Podaj a:");
7. scanf("%d", &a);
8. printf("\nPodaj b:");
9. scanf("%d", &b);
10. c=a+b;
```

```
11. printf("\n Suma %d + %d =%d", a,b,c);
12. c=a-b;
13. printf("\n Roznica %d - %d =%d", a,b,c);
14.  c=a*b;
15. printf("\n Iloczyn %d * %d =%d", a,b,c);
16. c=a/b;
17. printf("\n Iloraz calkowity  %d  przez %d = %d", a,b,c);
18. c=a%b;
19. printf("\n Reszta z dzielenia całkowitego %d / %d = %d ",a, b, c);
20. getch();
21. return 0;
22. }
```

Przykładowy przebieg programu.

Podaj a:12

Podaj b:5

Suma 12 + 5 =17

Roznica 12 - 5 =7

Iloczyn 12 * 5 =60

Iloraz calkowity 12 przez 5 = 2

Reszta z dzielenia calkowitego 12 / 5 = 2

Komentarz do przykładu 5

Linia 7 `scanf("%d", &a);`

Program może wymagać wprowadzenia pewnych wartości z zewnątrz, np. z klawiatury. Do realizacji tego zadania może być wykorzystana funkcja `scanf`. Funkcja powoduje wstrzymanie realizacji programu i oczekuje na wprowadzenie ciągu znaków reprezentującego liczbę, która ma być umieszczona w pewnej zmiennej. Wprowadzanie kończy się po podaniu ENTER. Ciąg znaków jest przekształcany na liczbę, o ile posiada odpowiednią postać, odpowiadającą formatowi wprowadzanej wartości. Pierwszy argument `"%d"` użytej tutaj funkcji `scanf` to tablica znaków zawierająca deskryptor formatu, wskazujący, że ma być wprowadzona liczba całkowita ze znakiem. Drugi argument jest adresem zmiennej (ściślej wskaźnikiem do zmiennej), do której ma być wprowadzona wartość. Znak & jest operatorem adresowym. Operator ten przyłożony do zmiennej po jej lewej stronie, daje adres tej zmiennej. Funkcja `scanf` dla formatu `%d`, pobierając znaki z bufora, odrzuca wszystkie początkowe białe znaki (ang, whitespaces). Pierwszy znak nieodpowiadający formatowi kończy pobieranie znaków. Jeśli pobrane znaki mogą reprezentować liczbę całkowitą, są przekształcane na liczbę, jeżeli nie, wartość zmiennej, do której miała zostać wprowadzona wartość, pozostaje niezmieniona.

Linia 8 `printf("\nPodaj b:");`

W linii tej pojawiła się sekwencja znaków \n zwana nową linią. Kombinacja ta ma specjalne znaczenie. Powoduje ona, że wyświetlanie kolejnych informacji wyprowadzanych przez program na ekran, będzie się odbywało od kolumny 1 w następnym wierszu. Sekwencja ta należy do tzw. sekwencji ucieczki (ang. escape sequence). Oznacza ona odejście od podstawowego znaczenia danej litery w łańcuchu znaków i utworzenie znaku sterującego. Inne sekwencje ucieczki przedstawiono w tabeli 3 (str. 29).

Linia 10 `c=a+b;`

Jeśli suma `a+b` mieści się w zakresie liczb całkowitych typu int (zakres ten podano na str.12, tabela 1, wiersz 6), wynik jest poprawny i przypisywany jest do zmiennej c. Natomiast, jeśli suma przekracza ten zakres, uzyskuje się wynik niepoprawny, ujemny lub dodatni. Wystąpienie znaku ujemnego wynika z faktu stosowania kodu U2, gdzie bit liczby o najwyższej wadze reprezentuje znak liczby i gdy wynik przekracza 31 bitów, pojawia się przeniesienie do pozycji znaku, co zmienia znak liczby. Podobne zachowanie może wystąpić dla odejmowania i mnożenia.

Poniżej w przykładzie 6 podano program realizujący te same działania, co program z przykładu 5, lecz dla arytmetyki liczb bez znaku reprezentowanych w naturalnym kodzie binarnym (NKB).

Przykład 6. *Program ilustrujący użycie operatorów arytmetyki całkowitej dla liczb bez znaku.*

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  int main(int argc, char* argv[])
4.  {
5.  unsigned int a,b,c;
6.  printf("Operatory arytmetyki dla typu unsigned int ");
7.  printf("\nPodaj a:");
8.   scanf("%u", &a);
9.  printf("\nPodaj b:");
10.  scanf("%u", &b);
11.  c=a+b;
12.  printf("\n Suma %u + %u = %u", a,b,c);
13.  c=a-b;
14.  printf("\n Roznica %u - %u = %u", a,b,c);
15.  c=a*b;
16.  printf("\n Iloczyn %u * %u =%u", a,b,c);
17.  c=a/b;
18.  printf("\n Iloraz calkowity  %u  / %u =%u", a,b,c);
19.  c=a%b;
20.  printf("\n Reszta z dzielenia całkowitego %u  / %u = %u",a,b,c);
21.  getch();
22.  return 0;
23.  }
```

Komentarz do przykładu 5

Linia 8 `scanf("%u", &a);`

Zastosowano tutaj deskryptor formatu %u służący do wprowadzania i wyprowadzania zmiennych bez znaku typu unsigned int, typ ten jest typem 32-bitowym o zakresie [0, $2^{32}-1$]. Jeśli wynik operacji arytmetycznej przekracza ten zakres, wynik jest obliczany modulo 2^{32} .

W kolejnym przykładzie zamieszczono obliczanie wyrażenia w arytmetyce całkowitej. Wyrażenie zawiera w liczniku obliczanie reszty i obliczanie wartości bezwzględnej przy zastosowaniu funkcji bibliotecznej abs. Należy tutaj zauważyć, że każda wykonywana tutaj operacja dzielenia jest dzieleniem całkowitym, przy którym pomijana jest część ułamkowa wyniku.

Przykład 7. *Program ilustrujący użycie operatorów arytmetyki całkowitej do obliczania wyrażenia*.

$$\frac{k^3 + |k|_5 + |l|}{k + \dfrac{k}{k+l}}$$

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main(int argc, char* argv[])
4. {
5. int k,l, licznik, mianownik, wynik;
6. printf("Podaj k:");
7. scanf("%d", &k);
8. printf("\n Podaj l:");
9. scanf("%d", &l);
10. licznik=k*k*k+ k%5 +abs(l);// funkcja abs oblicza moduł argumentu
11. mianownik=k+ k/(k+l);
12. wynik=licznik/mianownik;
13. printf("\n Wartosc wyrazenia=%d", wynik);
14. getch();
15. return 0;
16. }
```

Przykładowy przebieg programu.

```
Podaj k: 9

Podaj l: -3

Wartosc wyrazenia=73
```

Operatory arytmetyki rzeczywistej

Do operatorów arytmetyki rzeczywistej należą:

-operatory jednoargumentowe ‘+’ i ‘ –’
-operator dodawania ‘+’
-operator odejmowania ‘-’
-operator mnożenia ‘*’
-operator dzielenia ‘/’

Przed przejściem do wykorzystania operatorów arytmetyki rzeczywistej rozważymy kwestię deskryptorów stosowanych do wprowadzania i wyprowadzania zmiennych rzeczywistych typu float, double i long double. Dla zmiennych typu float do wprowadzania wartości stosuje się deskryptor % f, dla zmiennych typu double deskryptor %lf, a dla zmiennych typu long double, %Lf. Przy wydruku można stosować te same deskryptory, można również dla wartości typu double zastosować deskryptor %f.

Przykład 8. *Program ilustrujący wczytywanie i drukowanie zmiennych rzeczywistych typu* float, double i long double.

```
1.  #include <stdio.h>
2.  #include <comio.h>
3.  int main(int argc, char* argv[])
4.  {
5.  float x;
6.  double y;
7.  long double z;
8.  printf(”Podaj x:”);
9.  scanf(”%f”, &x); // %f jest deskryptorem formatu stosowanym przy wczytywaniu
10.                  // i drukowaniu zmiennych typu float
11. printf(”\nPodaj y:”);
12. scanf(”%lf”, &y);// %lf deskryptor formatu dla zmiennych typu double
13. printf(”\nPodaj z:”);
14. scanf(”%Lf”, &z);// %Lf deskryptor formatu dla zmiennych typu long double
15. printf(”\n x=%f”, x);
16. printf(”\n y=%f”, y);
17. printf(”\n z=%Lf”, z);
18. getch();
19. return 0;
20. }
```

Poniżej podano przykładowe przebiegi programu dla ułamków i liczb całkowitych

Kompilator DevC++ i kompilator Borland 6.0

```
Podaj x:0.12345678

 Podaj y:0.123456789123456

Podaj z:0.123456789123456789

 x=0.123457
 y=0.123457
 z=0.123457
```

Uzyskujemy tutaj wydruki z 6 cyframi po kropce.

Kompilator DevC++

```
Podaj x:12345678

Podaj y:123456789123456

Podaj z:1234567891234567890

 x=12345678.000000
 y=123456789123456.000000
 z=12345678912345678.000000
```

Kompilator Borland 6.0

```
Podaj x:12345678

Podaj y:123456789123456

Podaj z:1234567891234567890

 x=12345678.000000
 y=123456789123456.000000
 z=12345678912345678.000000
```

Warto tutaj zauważyć, że dla kompilatora DevC++ typ long double nie zachowuje się zgodnie ze standardem ANSI C, gdyż daje tylko 17 cyfr, zamiast 19-20.

Przykład 9. *Program ilustrujący użycie operatorów arytmetyki rzeczywistej.*

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  int main(int argc, char* argv[])
4.  {
5.  double a,b,c;
6.  printf(”Podaj a:”);
7.  scanf(”%lf”, &a);
8.  printf(”\nPodaj b:”);
9.  scanf(”%lf”, &b);
10. c=a+b;
11. printf(”\n Suma %f + %f =%f”, a,b,c);
12. c=a-b;
13. printf(”\n Roznica %f - %f =%10.2f”, a,b,c);
14. c=a*b;
15. printf(”\n Iloczyn %f * %f =%.4f”, a,b,c);
16. c=a/b;
17. printf(”\n Iloraz  %f  / %f =%f”, a,b,c);
18. getch();
```

```
19. return 0;
20. }
```

Przykładowy przebieg programu.

```
Podaj a:2.345

Podaj b:3.4325

 Suma 2.345000 + 3.432500 =5.777500
 Roznica 2.345000 - 3.432500 =     -1.09
 Iloczyn 2.345000 * 3.432500 =8.0492
 Iloraz   2.345000  / 3.432500 =0.683176
```

Komentarz do przykładu 9.

Linia 13 `printf("\n Roznica %f - %f =%10.2f", a,b,c);`

Do formatowania wydruku różnicy zastosowano opis formatu jako `%10.2f`. Format taki oznacza wydruk w polu o szerokości 10 z dwoma cyframi po kropce dziesiętnej.

Linia 15 `printf("\n Iloczyn %f * %f =%.4f", a,b,c);`

Do formatowania wydruku iloczynu zastosowano opis formatu jako `%0.4f`. Format taki określa, że na wydruku powinny być 4 cyfry po kropce dziesiętnej.

Z obliczaniem wyrażeń w języku C wiążą się trzy zagadnienia:

a) problem kolejności obliczania podwyrażeń

b) konwersja typów w wyrażeniach

c) rzutowanie (zmiana typu wyrażenia)

Ad a)

Język C nie określa kolejności, w jakiej są obliczane podwyrażenia danego wyrażenia. Kompilator ma więc swobodę w zmianie kolejności obliczania tak, aby uzyskać zoptymalizowany kod. Przykładowo w wyrażeniu po prawej stronie

```
x=f1() +f2();
```

nie można określić, czy funkcja `f1` będzie uruchomiona (wywołana) jako pierwsza.

Ad. b)

Jeżeli w wyrażeniu występują zmienne i stałe różnych typów, są one wszystkie przekształcane do jednego wspólnego typu. Kompilator przekształca wszystkie operandy do największego typu występującego w wyrażeniu. W pierwszym rzędzie wartości typu `char` i `short int` przekształcane są do typu `int`.

Przekształcenia typów przebiegają zgodnie z powyższymi regułami:

- operand jest typu long double, to drugi operand jest przekształcany do typu long double,
- operand jest typu double, to drugi operand jest przekształcany do typu double,
- operand jest typu unsigned long int, to drugi operand jest przekształcany do typu unsigned long int,
- operand jest typu float, to drugi operand jest przekształcany do typu float,
- operand jest typu long int, to drugi operand jest przekształcany do typu long int,
- operand jest typu unsigned int, to drugi operand jest przekształcany do typu unsigned int,

Istnieje jednak pewien specjalny przypadek: jeżeli jeden operand jest typu long int a drugi typu unsigned long int, i wartości typu unsigned long int nie można przedstawić przy użyciu typu long int, wtedy oba operandy są przekształcane do typu unsigned long int.

Po zastosowaniu powyższych reguł każda para operandów ma ten sam typ oraz wynik operacji jest tego samego typu, co typ obu operandów.

Ad.c.

Typ wyrażenia można narzucić lub zmienić przy pomocy tzw. rzutowania. Ogólna postać rzutowania to:

(typ) wyrażenie,

lub

typ (wyrażenie)

Rzutowanie jest operatorem o priorytecie takim jak pozostałe operatory jednoargumentowe.

Przykład 10. *Rzutowanie zmiennych typu* int *na typ* double.

Rozważmy fragment programu:

```
1. double wynikProc;
2. int iloscProb=100;
3. int iloscUdanychProb=35;
4. wynikProc= (double)iloscUdanychProb/ iloscProb*100;
```

Linia 4 `wynikProc= (double)iloscUdanychProb/ iloscProb*100;`

Zastosowano tutaj rzutowanie zmiennej iloscUdanychProb typu int na typ double. Powoduje to, że po prawej stronie wyrażenia zostanie użyty operator dzielenia rzeczywistego i wynikiem będzie liczba rzeczywista przypisywana do zmiennej wynikProc. Gdyby rzutowania nie było, zostałby użyty operator dzielenia całkowitego, co spowodowałoby wykonanie dzielenia całkowitego, którego wynikiem byłoby 0, gdyż

iloscUdanychProb< iloscProb.

co powoduje, że całkowita część ilorazu tych wartości jest równa 0.

Przykład 11. *Napisać fragment programu określający procentową ilość udanych prób w stosunku do ogólnej ilości prób(zmiana typu poprzez rzutowanie).*

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main(int argc, char* argv[])
4. {
5. int ilosc_prob, ilosc_udanych_prob;
6. double udzial_procentowy;
7. printf("\n Podaj ilosc prob :");
8. scanf("%d",&ilosc_prob);
9. printf("\n Podaj ilosc udanych prob:");
10. scanf("%d",&ilosc_udanych_prob);
11. udzial_procentowy= (double )ilosc_udanych_prob/ilosc_prob*100;
12. printf("\nUdzial procentowy  udanych prob=%6.2f", udzial_procentowy);
13. getch();
14. return 0;
15. }
```

Przykładowy przebieg programu.

```
Podaj ilosc prob :95

Podaj ilosc udanych prob:28

Udzial procentowy udanych prob= 29.47 %
```

4 Podstawy tworzenia łańcuchów formatujących

(pełny opis w B.W. Kernighan, D.M. Ritchie, Język ANSI C, Dodatek B, Pkt. B1.2 Formatowanie wyjścia)

Formatowanie wydruku oznacza tworzenie ciągu znaków reprezentującego wartość, która będzie wyprowadzana na urządzenie zewnętrzne (monitor, drukarka) lub też zapamiętywana. Dla wartości każdego typu stosuje się odpowiedni sposób opisu tworzenia takiego ciągu. Sekwencję znaków opisującą tworzenie zewnętrznej postaci wyprowadzanej wartości nazywamy łańcuchem formatującym.

Do formatowania liczb całkowitych stosuje się deskryptory %d, %u, %o, %x, %X. Deskryptor %d służy do wyprowadzania liczb dziesiętnych całkowitych ze znakiem, %u liczb dziesiętnych całkowitych bez znaku, %o liczb ósemkowych bez znaku. Deskryptory %x i %X, służą do wyprowadzania liczb szesnastkowych bez znaku, odpowiednio z użyciem małych liter i dużych liter.

Przykład 12. *Wydruk liczby całkowitej w postaci dziesiętnej, oktalnej i heksadecymalnej.*

```
1. int l=125;
2. printf("\n l dec=%d l oct=%o l hex=%x", l, l ,l );
```

Przy uruchomieniu tego fragmentu uzyskujemy

```
l dec=125 l oct=175 l hex=7d
```

Do formatowania liczb rzeczywistych stosuje się deskryptory %f, %lf, ,%Lf , %e, %E, %g, %G.. Funkcjonują one w sposób następujący:

- %f - deskryptor dla typu float z precyzją 6 cyfr znaczących
- %lf - deskryptor dla typu double
- %Lf - deskryptor dla typu long double
- %e, %E - deskryptory dla wartości rzeczywistych dające wydruk w tzw. postaci naukowej *m*.*dddddd***e** xx lub *m*.*dddddd***E** xx , **e** lub **E** symbolizują wykładnik 10.
- %g - deskryptor ten powoduje wybór %f lub %e, w zależności od tego, który daje krótszy zapis

Przykład 13.*Przykładowe formatowanie wydruku zmiennej typu* double.

```
1. double x=12.348;
2. printf("\n x=%8.2lf", x);
```

W wyniku uruchomienia tego fragmentu programu otrzymujemy

```
x=   12.35
```

Opis formatu rozpoczyna się znakiem %, liczba 8 oznacza minimalną szerokość pola wydruku, liczba 2 ilość drukowanych miejsc po kropce dziesiętnej, a lf oznacza wydruk wartości typu double. W ogólnym przypadku pomiędzy znakiem % a znakiem przekształcenia mogą wystąpić modyfikatory (umieszczane w dowolnej kolejności), które wpływają na postać wydruku:
Modyfikatory te to:

- znak minus (-) powodujący dosunięcie wydruku do lewego krańca pola
- znak plus (+) wypisanie liczby zawsze ze znakiem
- *odstęp* poprzedzenie wyniku znakiem odstępu, jeżeli pierwszym drukowanym znakiem nie jest plus lub minus
- znak zero (0) uzupełnienie liczby wiodącymi zerami do pełnego rozmiaru pola
- litera l lub L, dla wydruku wartości typu double lub long double.

Następnym znakiem jest **liczba określająca minimalny rozmiar pola**. Przekształcony argument zostanie wydrukowany w polu o ilości znaków nie mniejszej niż ta liczba. Jeżeli argument wymaga szerszego pola, to pole wydruku jest rozszerzane. Jeżeli argument jest krótszy, dopełniany jest

spacjami lub zerami w zależności od modyfikatora.

Kolejnym znakiem jest **kropka**, po której następuje liczba oznaczająca precyzję, czyli ilość cyfr po kropce dziesiętnej w specyfikacjach f, e i E. Dla deskryptorów tych domyślną precyzją jest 6, przy precyzji zero opuszcza się kropkę dziesiętną.

Przykład 14. *Formaty z zastosowaniem modyfikatorów.*

```
1. x=25.3485;
2. printf("\n x=% 020.4f",x);// uzupełnienie zerami
3. printf("\n x=% +20.4f",x)// drukowanie znaku liczby
4. printf("\n x=% -20.4f",x);//dosunięcie do lewego krańca pola
```

Otrzymane wydruki:

```
x= 000000000000025.3485
x=                +25.3485
x= 25.3485
```

5 Wartości logiczne, operatory relacji i operatory logiczne

-Wartość logiczna **prawda** jest reprezentowana przez każdą liczbę różną od zera, np. 1,2, -5.

-Wartość logiczna **fałsz** jest reprezentowana przez 0.

Operatory relacyjne

Operatory te to:

' < '	-mniejsze
'< ='	-mniejsze lub równe
'>' -	-większe
' >='	-większe lub równe
'=='	-równe
'!='	-różne

Wyrażenie relacyjne to wyrażenie składające się z operandów połączonych pewnym operatorem relacji

op1 **operator_relacyjny** op2.

Przykładami wyrażeń relacyjnych mogą być: a>b, a!=b, a==b.

Wartością wyrażenia relacyjnego jest 1, jeżeli wyrażenie jest prawdziwe albo też 0, gdy wyrażenie jest fałszywe.

Przykład 15. Prosty program ilustrujący zastosowanie operatorów relacyjnych.

```
#include <stdio.h>
#include <conio.h.>
 int main(int argc, char * argv[])
 {
 int a,b,c;
printf(" Podaj a=:");
scanf ("%d", &a);
printf(" Podaj b=:");
scanf ("%d", &b);
c=a<b;
printf ("\n Wartosc wyrazenia %d < %d=%d",a,b,c);
c=a<=b;
printf ("\n Wartosc wyrazenia %d <= %d=%d",a,b,c);
c=a==b;
printf ("\n Wartosc wyrazenia %d ==%d=%d",a,b,c);
c=a>b;
printf ("\n Wartosc wyrazenia %d > %d=%d",a,b,c);
c=a>=b;
printf ("\n Wartosc wyrazenia %d > =%d=%d",a,b,c);
c=a!=b;
printf ("\n Wartosc wyrazenia %d != %d=%d",a,b,c);
getch();
return 0;
}
```

Przykładowy przebieg programu.

```
Podaj a=1
Podaj b=0
Wartosc wyrazenia 1 < 0=0
 Wartosc wyrazenia 1 <= 0=0
 Wartosc wyrazenia 1 ==0=0
 Wartosc wyrazenia 1 > 0=1
 Wartosc wyrazenia 1 > =0=1
 Wartosc wyrazenia 1 != 0=1
```

Operatory logiczne w języku C

Operatory te to:

! -	- operator negacji
&&	- operator koniunkcji(iloczynu logicznego)
||	- operator alternatywy (sumy logicznej)

Poniżej w Tabeli 3 zamieszczono wyniki zastosowania poszczególnych operatorów logicznych.

Tabela 3. Tabela prawdy dla operatorów logicznych

x	*y*	*x&&y*	*x||y*	*!x*
0(fałsz)	0	0	0	1
0(fałsz)	0	0	1	1
1(prawda)	0	0	1	0
1(prawda)	0	1	1	0

Przykład 16. *Program ilustrujący najprostsze użycie operatorów logicznych.*

```
1.  #include <stdio.h>
2.  #include <conio.h.>
3.  int main(int argc, char * arg[])
4.  {
5.  int a,b,c;
6.  printf("\nPodaj a=:");
7.  scanf ("%d", &a);
8.  printf("\nPodaj b=:");
9.  scanf (”%d”, &b);
10. c=a&&b;
11. printf ("\n Wartosc wyrazenia %d  && %d = %d",a,b,c);
12. c=a||b;
13. printf ("\n Wartosc wyrazenia %d || %d = %d",a,b,c);
14. c=!a;
15. printf ("\n Wartosc wyrazenia  !%d = %d",a,c);
16. getch();
17. return 0;
18. }
```

Przykładowe wyniki działania programu.

```
Podaj a=3

Podaj b=0

Wartosc wyrazenia 3 && 0 = 0

Wartosc wyrazenia 3 || 0 = 1

Wartosc wyrazenia !3 = 0
```

Komentarz do przykładu 16.

Linia 7 `scanf ("%d", &a);`

W linii tej i w **linii 9** wprowadzane są dwie liczby całkowite. W sytuacji zastosowania tych wartości jako operandów operatorów logicznych w **liniach 11**, **12** i **14** są one traktowane jako wartości logiczne. Wprowadzane liczby różne od zera są traktowane jako wartość logiczna prawda, a wartość 0 jako wartość logiczna fałsz.

Wyrażenie logiczne jest kombinacją wyrażeń relacyjnych i logicznych. Obliczanie wyrażenia logicznego odbywa się przy zastosowaniu zasad pierwszeństwa operatorów, ponadto przy obliczaniu iloczynu logicznego, jeśli pierwszy argument operatora ma wartość logiczną fałsz(0), drugi argument nie jest obliczany. Podobnie dla operatora ||, jeśli pierwszy argument operatora ma wartość logiczną prawda (1), drugi argument nie jest obliczany.

Przykład 17. *Wyrażenia logiczne. Dla obliczenia wartości w poniższych wyrażeniach przyjęto* `x=1`, `y=4`, `z=1`.

Wyrażenie	***Wartość wyrażenia***
x<=1 && y==3	0
x<=1||y==3	1
!(x>1)	1
!x>1	0
!(x<=1||y==3)	0

Priorytety wybranych operatorów

O kolejności stosowania operatorów w wyrażeniu decyduje ich priorytet (pierwszeństwo). Priorytet operatora decyduje o tym, czy będzie on użyty przed innymi operatorami, czy też po nich. Przykładowo, w wyrażeniu

```
a+b*c,
```

stosowany jest najpierw operator * , a dopiero później operator +. Podobnie w wyrażeniu

```
a>b && c<d || e>f,
```

stosowane są najpierw operatory relacyjne, a później operator **&&** i następnie operator ||.

Przykładowo, jeśli mamy przykładową definicję zmiennych

```
int a=2, b=3, c=4, d=5, e=6, f=8, wynik;
```

to gdy użyjemy je do realizacji wyżej przedstawionego wyrażenia logicznego, mamy następujące obliczenie

```
wynik=2>3 && 4<5 ||  6>8;
```

Pierwsze wyrażenie relacyjne 2>3 ma wartość logiczną fałsz, w związku z tym określa ono wartość iloczynu logicznego i ma on wartość logiczną fałsz (jeśli pierwszy człon wyrażenia logicznego wyznacza jego wartość, drugi człon nie jest obliczany), wyrażenie relacyjne 6>8 ma

wartość fałsz, więc wartością całego wyrażenia jest wartość logiczna fałsz, a zmienna wynik ma wartość 0.

Hierarchia operatorów (dotychczas stosowanych)

Poziom 1 (najwyższy)	` −′ , `**+**′ (jednoargumentowy), `**!**′
Poziom 2	` ***′, `**/**′, `%′
Poziom 3	`**+**′, `**-**′
Poziom 4	`**<**`, `**<=**`, `**>**′,′**>=**`,
Poziom 5	`**==**`, `**!=**`
Poziom 6	`&&**′**
Poziom 7	` ||′
Poziom 8	`**=**`

Miejsce operatora w hierarchii określa jego priorytet (pierwszeństwo) podczas stosowania operatorów do obliczania wyrażenia. Operatory o wyższym priorytecie są stosowane przed operatorami o niższym priorytecie. W przypadku operatorów o tym samym priorytecie, kolejność stosowania określa tzw. łączność operatora, która może być prawostronna lub lewostronna. Zagadnienie to jak i pełna hierarchia operatorów są przedstawione np. w [2], [3], [8].

6 Identyfikatory i typ znakowy

Pisząc program musimy wybrać nazwy dla zmiennych, funkcji i innych obiektów. Nazwy te określane są jako identyfikatory.

Cechy identyfikatora to:

- Identyfikator(nazwa) jest sekwencją liter i cyfr o dowolnej długości (niektóre kompilatory wprowadzają tu pewne ograniczenia)
- Pierwszy znak identyfikatora musi być literą, przy czym znak podkreślenia '_' jest traktowany jako litera.
- Litery duże i małe są rozróżniane.
- Służą one do zapisu konstrukcji, jakie są dopuszczalne w C.
- Identyfikator nie może być **słowem kluczowym** (niektóre identyfikatory zostały zastrzeżone przez twórców języka C, zostaną one wymienione poniżej).

Budowa identyfikatora

Identyfikator w C musi spełniać następujące wymagania:

a) zaczynać od litery (A do Z, a do z) lub też od znaku podkreślenia _

b) składać z liter (A do Z, a do z), cyfr lub też znaków podkreślenia _

c) nie może być słowem kluczowym (słowo kluczowe to takie słowo jak for czy while).

W identyfikatorach nie można stosować polskich znaków.

Przykład 18. *Legalne (dopuszczalne) identyfikatory.*

```
ilosc
ilosc_calkowita
 _suma
kolumna3
ILOSC
```

Z wyjątkiem identyfikatorów zewnętrznych (nazw zewnętrznych), pierwsze 31 znaków każdego identyfikatora to tzw. znaki znaczące. Jest to wymaganie minimalne, aby dany kompilator był zgodny z normą ANSI C. Jeżeli dwa identyfikatory różnią się od 32 znaku, mogą być uznane za ten sam identyfikator. Niektóre kompilatory dopuszczają znacznie większe długości identyfikatorów, np. kompilator Borland Builder 6.0 pozwalał na identyfikatory o długości do 250 znaków. Ze względu na to, że rozróżniane są litery małe i wielkie, w powyższym przykładzie identyfikatory ilosc i ILOSC są różnymi identyfikatorami.

Przykład 19. *Identyfikatory nielegalne (niepoprawne).*

```
ilosc$          (niedopuszczalny znak $)

2_suma          (rozpoczyna się cyfrą)

long            (słowo kluczowe)

druga suma      (w identyfikatorze nie może być spacji)

ILOSC-CALKOWITA   (niedopuszczalny znak -)
```

Słowa kluczowe

W języku C istnieją predefiniowane identyfikatory, zwane słowami kluczowymi, które mają specjalne znaczenie i mogą być stosowane w programie w sposób określony przez gramatykę języka C.

W języku ANSI C istnieją następujące słowa kluczowe:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Znaczenie poszczególnych słów kluczowych jest wyjaśniane w niniejszym opracowaniu w miarę pojawiania się konstrukcji języka wykorzystujących te słowa. Jednak ze względu na ograniczone ramy opracowania, nie będzie omawiane znaczenie słowa enum związanego z definiowaniem typu wyliczeniowego jak słowa union oznaczającego tzw. unię będąca specjalnym przypadkiem typu strukturowego. Objaśnienie tych wszystkich słów kluczowych można znaleźć w literaturze, np.[8].

Komentarze

Komentarze są ciągami znaków ujętych w pary znaki znaków /* */. Komentarze są ignorowane przez kompilator. W komentarzach można używać polskich znaków. Jeśli stosuje się kompilator C++ lub standardu C99, można też stosować komentarz jednowierszowy rozpoczynający się od znaków //. Komentarz taki rozciąga się do końca wiersza.

Przykład 20. *Komentarz wielowierszowy i komentarz jednowierszowy*

```
/* Komentarz wielowierszowy
        Taki komentarz w języku C
        może zawierać  opisy składające się
        z wielu wierszy
*/

// Ten komentarz jest komentarzem jednowierszowym.
```

Typ znakowy

W języku C istnieje specjalny typ char o zakresie [-128,127] przeznaczony do operacji na znakach. Zmienne i stałe tego typu zajmują 1 bajt. Zmienne są definiowane w sposób następujący:

```
char ch;
```

W języku C istnieje szereg funkcji istnieje umożliwiających wykonywanie operacji na znakach takich jak wprowadzanie znaków z klawiatury, wyprowadzanie znaków na ekran i klasyfikacja znaków.

Wejście znakowe

Znaki mogą być wczytywane z klawiatury przy zastosowaniu funkcji pochodzących ze standardowej biblioteki języka C lub też funkcji dostarczanych przez producentów kompilatorów.

1.Funkcja scanf.

Przy wczytywaniu znaków konieczne jest stosowanie deskryptora format *%c*.

```
scanf ("%c",&ch);
```

2. Funkcja getchar().

```
ch=getchar();
```

3. Funkcja getch()

```
ch=getch();
```

Wczytywanie znaku przy użyciu scanf i getchar wymaga Enter po wprowadzeniu znaku.

Funkcja getch() nie wyprowadza wczytanego znaku na ekranie.

Wyjście znakowe

Wyprowadzenie znaku na ekran może być zrealizowane przy zastosowaniu jednej z trzech funkcji:

a) printf("Znak=%c", ch);
b) putchar(ch);
c) putch (ch);

Poniższy program umożliwia drukowanie tabeli kodów ASCII, drukowany jest znak (o ile jest znakiem drukowalnym) oraz jego kod ASCII w postaci dziesiętnej, heksadecymalnej i oktalnej.

Przykład 21. Drukowanie wybranych znaków i ich kodów ASCII.

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  int main()
4.  {
5.   int  i ;
6.   i=32;// kod spacji, kod pierwszego drukowanego znaku
7.   system("cls"); // wywołanie funkcji czyszczenia ekranu
8.   while (i<=255)// jeśli kod znaku jest mniejszy lub równy 255, wykonywana jest
9.  // treść pętli while
10.  {
11.   printf("\n \n Znak =%c  kod  dec=%3d  hex =%x oct=%o", i , i, i, i );
12.   getch();
13.   i=i+1;
14.  }
15.  return 0;
16.  }
```

Przykładowy fragment wykonania programu.

```
Znak = ć  kod  dec=134  hex =86 oct=206
Znak = ç  kod   dec=135  hex =87 oct=207
Znak = ł  kod   dec=136  hex =88 oct=210
```

Jak widać z powyższego wydruku litera ć ma kod oktalny 206, a litera ł 210.

Program umożliwia ustalenie kodów polskich liter, które pozwalają na drukowanie napisów, w których występują polskie litery.

Stałe znakowe

Stała znakowa to ciąg złożony z jednego lub większej liczby znaków ujęty w pojedyncze apostrofy np. 'x'. Stałe jednoznakowe są typu char, a wieloznakowe typu int. Stałe znakowe mogą być używane w operacjach arytmetycznych. W operacjach stosowany jest kod ASCII liczby np.

```
int cyfra;
cyfra= '5'-'0';
```

Faktycznie jest to odejmowanie kodów znaków. Kod '5' to 53, a kod '0' to 48, więc jest wykonywane odejmowanie 53-48, czyli w zmiennej cyfra będzie wartość 5.

Stałe znakowe mogą być zapisywane również przy zastosowaniu kodów heksadecymalnych i oktalnych, np. stałą znakową, np. 'A' można zapisać także jako '\x41' albo też '\101'. Mechanizm ten ma szczególne znaczenie, jeżeli chce się drukować znaki, które nie występują na klawiaturze. Przykładowo, napis "Róża bez kolców" można wydrukować w sposób następujący:

```
1. printf("\n R\242\276a bez kolc\242w"); //wersja oktalna
2. printf("\n R\xa2\xBE a bez kolc\xA2w");// wersja heksadecymalna
```

W powyższych instrukcjach zastosowano kody oktalne i heksadecymalne liter ó i ż. 242 i A2 są kodami ó, a 276 i BE litery ż.

Sekwencje ucieczki (ang. escape sequences)

Znaki, które nie mają reprezentacji graficznej na ekranie monitora lub też mają specjalne znaczenie w łańcuchach znaków, mogą być wygenerowane przy użyciu tzw. sekwencji ucieczki złożonych z ukośnika \ i odpowiedniego znaku.

Tabela 4. Sekwencje ucieczki

Nazwa sekwencji	*Symbol*	*Zapis znakowy*	*Wartość liczbowa (dec)*
Nowa strona (form feed)	FF	\f	12
Dzwonek (alert)	BEL	\a	7
Lewy ukośnik	\	\\	92
Znak zapytania	?	\?	63
Pojedynczy apostrof	'	\'	39
Podwójny apostrof	"	\"	34
Znak zerowy	NUL	\0	0
Nowy wiersz	NL(LF)	\n	10
Tabulacja pozioma	HT	\t	9
Tabulacja pionowa	VT	\v	11
Kasowanie znaku (backspace)	BS	\b	8
Powrót karetki	CR	\r	13

7 Instrukcje sterujące

Instrukcja złożona (grupująca)

Instrukcja złożona składa się z nawiasu klamrowego otwierającego, dowolnych instrukcji (mogą być również kolejne instrukcje złożone) i nawiasu klamrowego zamykającego:

Przykład 22. *Instrukcja złożona (grupująca).*

```
1. {
2.  printf("Instrukcja 1" );
3.  printf("Instrukcja 2\n");
4. }
```

Wszystkie instrukcje w języku C z wyjątkiem instrukcji złożonej kończą się średnikiem. Instrukcja złożona jest stosowana wtedy, gdy istnieje konieczność zgrupowania kilku instrukcji, tak aby były traktowane jako jedna instrukcja.

Instrukcja while(pętla while)

Instrukcja while jest instrukcją iteracyjną, umożliwiającą powtarzanie pewnego ciągu operacji. Instrukcja while ma następującą postać:

while (wyrażenie)

akcja

W instrukcji while najpierw określa się czy wyrażenie ma wartość logiczną prawda czy też fałsz. C wymaga by wyrażenie było zawarte w nawiasach. Jeżeli wyrażenie ma wartość logiczną prawda, wykonywana jest akcja i następuje powrót do obliczania wyrażenia. wyrażenie jest ponownie testowane i jeżeli dalej jest prawdziwe, znowu wykonywana jest akcja i następuje kolejny powrót do obliczania wyrażenia. Jeżeli jednak przy którymś obliczaniu wyrażenia, stwierdzone zostanie, że wyrażenie ma wartość logiczną fałsz, następuje natychmiastowe przejście do instrukcji znajdującej się po instrukcji while.

Akcja może się składać z jednej instrukcji lub też szeregu instrukcji zawartych w nawiasach { }. Użycie tych nawiasów jest konieczne, jeśli akcja obejmuje więcej niż jedną instrukcję.

Przykład 23. *Instrukcja* while.

```
1. int x=0;
2. while (x != 2)
3. {
4.  x = x + 1;
5.  printf ("\n x =  %d", x);
6. }
```

Powyższy fragment programu działa w sposób następujący: najpierw zmienna x jest inicjalizowana wartością 0, w pierwszym kroku obliczane jest wyrażenie sterujące pętlą, którym jest wyrażenie relacyjne x **!=** 2 . Wyrażenie to ma wartość logiczną prawda, gdyż x=0, x jest zwiększany o 1, drukowany jest napis x=1 i sterowanie pętli przechodzi do ponownego obliczenia wyrażenia relacyjnego, x jest zwiększany o 1, i drukowany jest napis x=2. Sterowanie pętli przechodzi pomownie do wyrażenia relacyjnego x!=2, które dla x=2 ma wartość fałsz i pętla ulega zakończeniu.

W pętli zastosowano instrukcję złożoną { }. Gdyby tej instrukcji nie było, byłaby realizowana tylko jedna instrukcja po pętli, czyli x=x+1. Instrukcja printf **(**"\n x = %d"**,** x**)** byłaby poza petlą i wartość x byłaby drukowana po zakończeniu działania pętli. Posłużenie się instrukcją grupującą pozwala potraktować obie instrukcje z punktu widzenia pętli jako jedną.

Instrukcja do while (pętla do while)

Instrukcja do while jest podobna do instrukcji while, z tą różnicą, że wyrażenie kontrolujące pętlę jest na końcu pętli. Cała pętla wykonywana jest przynajmniej raz.

```
do
        akcja
while (wyrażenie);
```

Przy wykonywaniu pętli do while wykonywana jest najpierw akcja, a następnie wykonywane jest sprawdzenie, czy wyrażenie wartość logiczną prawda też nie. Jeżeli ma wartość prawda, następuje powrót do początku pętli. Jeżeli w którymś momencie wyrażenie przybiera wartość fałsz, wykonywana jest pierwsza instrukcja występująca po pętli. Część pętli oznaczona jako akcja może być jedną instrukcją lub też grupą instrukcji zamkniętą w nawiasy { }, czyli instrukcją złożoną.

Instrukcja do while jest użyteczna wtedy, gdy test kontynuacji pętli w sposób naturalny znajduje się na końcu pętli. Przykładem takiej sytuacji może być weryfikacja formatu wartości wprowadzanej przez użytkownika z klawiatury. Po wprowadzeniu ciągu znaków odbywa się sprawdzanie formatu. Jeżeli format jest niedopuszczalny, użytkownik zachęcany jest do wprowadzenia innego ciągu znaków albo też, jeśli format jest poprawny, a podana wartość jest nieprawidłowa, wprowadzenia innej wartości.

Przykład 24. *Użycie instrukcji* do while *do wprowadzenia liczby dodatniej ze sprawdzeniem formatu.*

```
1. #include<stdio.h>
2. #include<conio.h>
3. int main()
4. {
5. int wartosc, k;
6. do
7. {
8.  printf ("\nPodaj liczbe calkowita:");
9.  k = scanf ("%d", &wartosc);
```

```
10.  fflush(stdin); //Opróżnianie standardowego strumienia wejściowego
11.  }
12.  while (k==0  || wartosc <=0);
13. printf("\n wprowadzona wartosc to: %d",wartosc);
14. getch();
15. return 0;
16.}
```

Wynik przykładowego uruchomienia programu.

```
Podaj liczbe calkowita > 0:A

Podaj liczbe calkowita > 0:-10

Podaj liczbe calkowita > 0:12

wprowadzona wartosc to: 12
```

Program nie pozwala na zakończenie wprowadzania danych bez podania dopuszczalnej wartości. Przy pierwszej próbie wprowadzenia wartości dla celów testowych podano literę A, która nie jest odpowiednia dla formatu liczby typu int. Przy drugiej próbie podano wartość -10, o poprawnym formacie, jednak ujemną, dopiero przy trzeciej próbie podano wartość 12, odpowiednią z punktu widzenia formatu i będącą liczbą dodatnią.

Komentarz do przykładu 24.

Linia 9 `k = scanf ("%d", &wartosc);`

Kod w tej linii ma kluczowe znaczenie dla działania pętli i całego procesu wprowadzania wartości. Funkcja scanf powoduje zatrzymanie wykonania programu i oczekuje na podanie ciągu znaków mogącego reprezentować liczbę całkowitą i zakończenie wprowadzania poprzez ENTER. Jeśli wprowadzony ciąg, po pominięciu początkowych białych znaków (np. spacji i tabulacji), może być przekształcony na liczbę całkowitą, to takie przekształcenie jest wykonywane i otrzymana liczba jest zapisywana do zmiennej wartość. Funkcja scanf zwraca wartość równą 1 i wartość ta przypisywana jest zmiennej k. Jeśli przekształcenie nie jest możliwe, funkcja zwraca wartość 0 i k przybiera wartość 0.

Linia 12 `while (k==0 || wartosc <=0);`

W linii tej obliczane jest wyrażenie logiczne z operatorem sumy logicznej. Jeśli ma ono wartość logiczną prawda, pętla jest kontynuowana. Wyrażenie ma wartość prawda, gdy prawdziwe jest przynajmniej jedno z wyrażeń relacyjnych. Pierwsze wyrażenie relacyjne k==0 ma wartość prawda, gdy k jest równe 0, czyli wtedy, gdy nie udało się wprowadzić wartości o odpowiednim formacie. Drugie wyrażenie relacyjne wartość<=0 ma wartość prawda, wtedy gdy wprowadzona liczba jest ujemna lub równa 0.

Instrukcja warunkowa if

Instrukcja warunkowa if umożliwia uzależnienie wykonania instrukcji od spełnienia pewnego warunku. Warunek ten jest reprezentowany przez wyrażenie umieszczone po słowie if. Wyrażenie

to przybiera wartość logiczną prawda lub wartość fałsz (wszystkie wartości różne od 0 są w języku C traktowane jako prawda, wartość 0 jako fałsz). Instrukcja ta ma następującą postać:

```
if (wyrażenie)
    akcja;
```

Wyrażenie musi być zawarte w nawiasach. Aby wykonać if, najpierw oblicza się wyrażenie i określa się czy jest **prawdziwe** czy też **fałszywe**. Jeżeli wyrażenie jest **prawdziwe**, zostaje wykonana akcja i sterowanie przechodzi do instrukcji programu umieszczonej po akcji. Jeżeli wyrażenie jest fałszywe, akcja zostaje pominięta i następuje przejście do instrukcji umieszczonej po akcji. Część if określona jako akcja składa się z pojedynczej instrukcji lub instrukcji złożonej (pewnej liczby instrukcji umieszczonych w nawiasach {}).

Przykład 25. *Instrukcja* if.

```
1.  int kod;
2.  printf("\n Podaj kod:");
3.  scanf("%d",&kod);
4.  if (kod= =1)
5.  printf ("\n warunek jest  spełniony");
```

Jeżeli z klawiatury podano wartość 1, fragment programu drukuje napis

```
warunek jest  spełniony
```

Nie należy mylić znaku = z parą znaków = =. Można wprawdzie napisać if (x=1), jednak wyrażenie po if jest tutaj równe, niezależnie od wartości x, więc instrukcja if nie będzie pełnić swojej roli.

Instrukcja if else

Instrukcja ta ma następującą postać:

```
if (wyrażenie)
        akcja1
else
        akcja2
```

Podobnie jak dla instrukcji if, najpierw obliczane jest wyrażenie po if. Jeżeli wyrażenie ma wartość logiczną prawda, wykonywana jest akcja1, natomiast jeżeli wyrażenie ma wartość fałsz, jest wykonywana akcja2.

Przykład 26. *Instrukcja* if else.

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  int main ()
4.  {
5.   int kod=3245;
6.   printf("\nPodaj kod");
```

```
7.  scanf(" %d",&kod);
8.  if (kod==3245)
9.  printf("\nPodano prawidlowy kod");
10.  else
11.  printf("\nKod nie jest poprawny");
12.   getch();
13.  return 0;
14.  }
```

Przykładowe przebiegi programu.

Przebieg 1.

```
Podaj kod: 2345
Kod nie jest poprawny
```

Przebieg 2

```
Podaj kod: 3245
Podano prawidlowy kod.
```

Przykład 27. *Instrukcja* if else *ze złożonymi instrukcjami po* if *i po* else.

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  int main()
4.  {
5.   int kod;
6.   printf("\n Podaj kod");
7.   scanf(" %d",&kod);
8.   if (kod!=2)
9.   {
10.    printf("\n Instrukcje wykonywane  ");
11.    printf("\n kod jest rozny od 2");
12.   }
13.   else
14.   {
15.   printf("\n Instrukcje wykonywane ");
16.   printf("\n kod jest rowny 2 ");
17.   }
18.    return 0;
19.   }
```

Przebieg 1.

```
Podaj kod: 2
Instrukcje wykonywane
 gdy kod jest rowny 2
```

Przebieg 2.

```
Podaj kod: 5
Instrukcje wykonywane
 gdy kod jest rozny od 2
```

Przykład 28. *Określić napisy drukowane przez poniższy fragment programu dla zmiennej* kod=2, 3 i 5.

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  int main()
4.  {
5.   int kod;
6.   printf("\n Podaj kod");
7.  scanf("%d",&kod);
8.   if (kod==2||k==3)
9.   {
10.   printf("\n Matematyk    ");
11.    if (kod==2)
12.  printf("artysta");
13.  else
14.  printf("programista");
15.  }
16.   else
17.   printf(”\nSpecjalista programista”);
18.   getch();
19.  return 0;
20.  }
```

Przebieg 1.

```
Podaj kod: 2
Matematyk  artysta
```

Przebieg 2.

```
Podaj kod: 3
Matematyk  programista
```

Przebieg 3.

```
Podaj kod: 5
Specjalista programista
```

Instrukcja if zagnieżdżone

W instrukcji takiej po if lub po else występuje kolejna instrukcja warunkowa if lub if else. Instrukcja taka ma budowę następującą:

```
if (wyrażenie1)
        akcja1;
else if (wyrażenie2)
        akcja2;
else if (wyrażenie3)
.
.
.
else if (wyrażenie n)
        akcja n;
```

Działanie instrukcji if zagnieżdżone

Instrukcja taka funkcjonuje w sposób następujący:

- jeżeli prawdziwy jest warunek pierwszy (wyrażenie1), to wykonywana jest akcja1 i następuje przejście do instrukcji umieszczonej po *akcji* n-tej.
- ogólnie, jeżeli jest prawdziwe wyrażenie i-te, to jest wykonywana i-ta akcja i następuje przejście do instrukcji po wyrażeniu n-tym.
- jest wykonywana tylko jedna akcja (grupa akcji w nawiasie klamrowym), pozostałe akcje są odrzucane.

Przykład 29. *Instrukcja* if *zagnieżdżone*.

```
1. #include<stdio.h>
2. #include <conio.h>
3. int main()
4. {
5.  int kod;
6.  printf("\nPodaj kod:");
7.  scanf("%d",&kod);
8.  if (kod= =1)
9.  printf("Akcja ta jest wykonywana gdy kod jest równy 1");/*instrukcja_1*/
10.  else if (kod= =2)
11.  printf("Akcja ta jest wykonywana gdy kod jest równy 2");/*instrukcja_2*/
12.  else if (kod<=3)
```

```
13.  printf("Akcja ta jest wykonywana, gdy kod jest mniejszy równy 3");/*instrukcja_3*/
14.  getch();
15.  return 0;
16.  }
```

Przebieg 1.

```
Podaj kod:1
Akcja ta jest wykonywana gdy kod jest rowny 1
```

Przebieg 2.

```
Podaj kod:2
Akcja ta jest wykonywana gdy kod jest rowny 2
```

Przebieg 3.

```
Podaj kod:0
Akcja ta jest wykonywana, gdy kod jest mniejszy lub rowny 3
```

Warto zauważyć, że takim sformułowaniu jak w przykładzie 29, kod związany z warunkiem kod<=3, nie jest w każdym przypadku wykonywany, pomimo że warunek jest spełniony. Warunek ten obejmuje też sytuacje kod=0 i kod=1, a wtedy są wykonywane wcześniejsze instrukcje i sterowanie nie dochodzi do sprawdzenia warunku kod<=3, .

Instrukcja if else z akcją domyślną

```
if (wyrażenie1)
      akcja1;
else if (wyrażenie2)
      akcja2;
else if (wyrażenie3)
.
.
.
else if (wyrażenieN)
      akcjaN;
else
akcja;
```

akcja jest tutaj działaniem domyślnym, które jest wykonywane, gdy żadna z akcji od 1 do N nie zostanie wykonana.

Przykład 30. *Instrukcja* if else *z akcją domyślną.*

```
1. #include<stdio.h>
2. #include <conio.h>
3. int main()
4. {
5.  int kod;
6.  printf("\nPodaj kod :" );
7.  scanf("%d",&kod);
8.  if (kod==1)
9.   printf("Akcja ta jest wykonywana, gdy kod jest rowny 1");
10.  else
11.  if (kod==2)
12.  printf("Akcja ta jest wykonywana, gdy kod jest rowny 2");
13.  else
14.  if (kod<=3)
15.  printf("Akcja ta jest wykonywana, gdy kod jest mniejszy lub rowny 3");
16.  else
17.  printf("Akcja ta jest wykonywana, gdy "
18.        "\nzadna z akcji 1, 2, 3 nie wykona sie");
19. getch();
20. return 0;
21. }
```

Przebieg 1.

```
 Podaj kod:2
Akcja ta jest wykonywana, gdy kod jest rowny 2
```

Przebieg 2.

```
Podaj kod:5

Akcja ta jest wykonywana, gdy
zadna z akcji 1, 2, 3 nie wykona sie
```

Instrukcja switch

Instrukcja switch jest instrukcją wyboru wielowariantowego, jest stosowana, gdy warunki wyboru mają wartość całkowitą, instrukcja ta może zastąpić wielokrotne if else.

```
switch( wyrazenieCalkowite)
{
case etykieta_1: instrukcja1;
case etykieta_2: instrukcja2;
.
.
.
case etykieta_n: instrukcjaN;
```

```
default: instrukcje;
}
```

Wyrażenie po słowie switch, zwane wyrażeniem wyboru lub selektorem, musi przyjmować wartości całkowite. Etykiety są całkowitymi wartościami stałymi lub wyrażeniami stałymi. Jeśli podczas wykonywania instrukcji switch jedna z etykiet ma wartość równą wartości selektora (oznaczonego jako wyrazenieCalkowite), to wykonanie instrukcji switch rozpoczyna się od wykonania instrukcji znajdującej się przy tej etykiecie.

Jeżeli w instrukcji switch występuje słowo default (ang. domyślny), instrukcje umieszczone po słowie default są wykonywane, gdy żadna z etykiet nie ma wartości równej selektorowi. Etykieta default jest opcjonalna i jeżeli nie występuje, to następuje przejście do pierwszej instrukcji po instrukcji switch.

Jeżeli chce się uzyskać wykonanie tylko jednej konkretnej instrukcji, należy po każdym wariancie umieścić słowo break wraz ze średnikiem. Spowoduje to natychmiastowe przerwanie wykonywania instrukcji switch po zrealizowaniu instrukcji związanej z danym wariantem.

Przykład 31. *W przykładzie wczytywana jest liczba całkowita i jeżeli jest to liczba 0,1,2 lub 3, drukowany jest komunikat informujący o wartości liczby. Jeżeli jest podaną liczbą jest inna liczba, drukowany jest komunikat* Inna liczba.

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  int main(int argc, char * argv{})
4.  {
5.  int liczba;
6.  printf("\n Podaj liczbe:");
7.  scanf("%d", &liczba);
8.  switch( liczba)
9.  {
10.  case 0: printf("\n Liczba 0");
11.  break;
12.  case 1: printf("\n Liczba 1");
13.  break;
14.  case 2:case 3: printf("\n Liczba 2 lub 3");
15.  break;
16.  default : printf("\n Inna liczba");
17.  }
18.  getch();
19.  return 0;
20.  }
```

Instrukcja for

Instrukcja for, zwana też pętlą for, jest instrukcją iteracyjną podobnie jak instrukcje while i do while. Stosowana jest ona do wielokrotnego powtarzania pewnego segmentu kodu, głównie wtedy gdy ilość powtórzeń jest znana.

Pętla for ma następującą postać:

```
for (wyr1; wyr2; wyr3)
        akcja
```

Wyrażenie wyr1 jest stosowane do inicjalizacji pętli, wyrażenie wyr2 jest używane do testowania warunku kontynuacji pętli, wyrażenie wyr3 służy do uaktualniania pętli. akcja stanowi treść (ciało) pętli.
Działanie:

Krok1. Obliczenie wyrażenia wyr1 (jest ono obliczane jeden raz i tylko jeden raz na początku działania pętli).

Krok 2a. Obliczenie wyr2 i sprawdzenie czy wyrażenie wyr2 jest prawdziwe czy też fałszywe.

Krok 2b. Jeżeli wyr2 jest fałszywe, zakończenie pętli i przejście do instrukcji umieszczonej bezpośrednio po pętli. Jeżeli wyr2 jest prawdziwe, wykonanie akcji.

Krok 3. Obliczenie wyrażenia wyr3, następnie przejście do **Kroku 2**.

Ciało pętli (akcja) może być pojedynczą instrukcją albo też instrukcją złożoną (grupującą), jeśli jest konieczne wykonanie w pętli więcej niż jednej instrukcji.

Przykład 32. *Użycie pętli* for.

```
1. int sum=0;
2. int i;
3. for (i=1; i<=4; i=i+1)
4. sum=sum+i;
```

W powyższym przykładzie w pętli for najpierw jest wykonywana instrukcja i=1, następnie sprawdzany jest warunek i<=4, dla i=1 jest on prawdziwy, więc wykonywana jest instrukcja sum=sum+i; i następuje przejście do instrukcji i=i+1, zmienna i przyjmuje wartość 2, i następnie wykonywane jest sprawdzenie i<=4, warunek ma wartość logiczną prawda, więc ponownie wykonywana jest instrukcja sum=sum+i. Gdy i przybierze wartość 5, warunek i<=4 nie spełniony i pętla kończy się.

Przykład 33. *Użycie pętli* for.(typowy zapis)

```
1. #include<stdio.h>
2. #include<conio.h>
3. int main(int argc, char* argv[])
4. {
```

```
5.  int i,sum;
6.  for( i=0,sum=0; i<=4;i++)/*zapis i=0.sum=0 oznacza tutaj wykonanie dwóch
7.  instrukcji podstawienia, jest to tzw. operator przecinkowy, szczegółowo
8.  omawiany w pkt. 1.8.  i++ oznacza i=i+1, ++ jest to operator
    inkrementacji, omawiany też w pkt.1.8 */
9.  sum=sum+i;
10.  printf("\n sum=%d",sum);
11.  getch();
12.  return 0;
13.  }
```

Pętla for może mieć kilka odmian.

1.Pętla nieskończona

```
for ( ; ; )
printf ("Pętla ta będzie działać w nieskończoność");
```

Brak wyr2 jest interpretowany jako wartość logiczna prawda.

2. Pętla bez treści(ciała)

```
for (i=0;i<100000;i++) ; // pętla taka może realizować opóźnienie w  programie
```

Ogólnie w pętli for może nie być któregoś z wyrażeń albo też wszystkich, jednak zawsze muszą być dwa średniki.

3**.** *Pętla bez wyrażenia* wyr3.

```
1.  for( i=0,sum=0; i<=4;)
2.  {
3.   sum=sum+i;
4.  i++;
5.  }
```

Instrukcja goto

Instrukcja goto to tzw. instrukcja skoku powodująca przekazanie bezwarunkowe przekazanie sterowania do punktu w programie opatrzonego etykietą. Etykieta jest identyfikatorem i zakończona jest dwukropkiem. Etykietę można umieścić przed każdą instrukcją w funkcji, w której występuje instrukcja goto.

Przykład 34. *Instrukcja* goto.

```
1.  int j=1;
```

```
2. E1: j++;
3. if (j<=100) goto E1;
4. printf(”\n j=%d”, j);
```

Stosowanie instrukcji goto nie jest zalecane, gdyż zwykle jej wielokrotne użycie powoduje, iż program staje się mniej czytelny jak również trudniejszy do modyfikacji. Istnieją jednak sytuacje, gdzie instrukcja goto jest wygodnym rozwiązaniem.

Przykład 35. *Użycie instrukcji* goto *do opuszczenia pętli zagłębionej*.

```
5. int i,j,k;
6. for ( i=0;i<100;i++)
7.  for (j=0;j<100;j++)
8.   for (k=0;k<50;k++)
9.    if (i+j+k>10000) goto E2;
10.    E2: printf (”\n Instrukcja poza petlami”);
```

Instrukcja break

Instrukcja break powoduje natychmiastowe przerwanie wykonywania pętli, w której została umieszczona.

Przykład 36. *Użycie instrukcji* break *w pętli*.

```
1. for (i=0;i<5;i++)
2. {
3. printf("\n i=%d ",i);
4. if (i==3) break; printf("\n Koncowa instrukcja petli");
5. }
6. printf("\n Poza petla");
```

W przykładzie powyższym, gdy zmienna i osiągnie wartość 3, warunek po if ma wartość logiczną prawda, co powoduje wykonanie instrukcji break, która daje natychmiastowe przerwanie wykonywania pętli.

Instrukcja continue

Instrukcja continue jest podobna do instrukcji break w tym sensie, że również przerywa działanie pętli, jednak pętla nie ulega zakończeniu, lecz następuje przeskok do wykonania wyrażeń sterujących pętlą. Dla pętli for mamy

```
1. for (wyr1;wyr2;wyr3)
2.   {
```

```
3.      blok instrukcji (1)
4.     if (warunek) continue;
5.      blok instrukcji (2)
6.   }
```

Jeżeli warunek po if jest prawdziwy, to wykonanie continue powoduje natychmiastowe przejście, bez wykonania bloku instrukcji(2) do obliczenia wyrażenia wyr3 i następnie do obliczenia wyrażenia wyr2. Jeżeli wyrażenie wyr2 ma wartość prawda, wykonanie pętli jest kontynuowane. Dla pętli while i pętli do while następuje natychmiastowe przejście do obliczenia wyrażenia sterującego pętlą, czyli dla pętli while na początek pętli, a dla pętli do while na koniec pętli.

Przykład 37. *Program oblicza średnią liczb dodatnich podawanych na wejście, liczby ujemne są odrzucane przy użyciu instrukcji* continue.

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main( argc, char* argv[])
4. {
5. double x, suma=0;
6. int ilosc=0;
7. while ( printf ("\n Podaj liczbe :"), scanf("%lf" ,&x)!=0)
8. {
9. if (x<0.0) continue;
10. suma=suma+x;
11. ilosc++;
12. }
13. if (ilosc>0)
14. printf("\n Srednia=%f", suma/ilosc);
15. else
16. printf("\n Nie podano zadnych liczb dodatnich");
17. getch();
18. return 0;
19. }
```

Przykładowy przebieg programu.

Podaj liczbe :15

Podaj liczbe :25

Podaj liczbe :5

Podaj liczbe :A

Srednia=15.000000

Program zakończył się, gdy na wejście podano znak (literę 'A'), który nie może być częścią ciągu znaków reprezentującego liczbę w postaci dziesiętnej.

8 Operatory warunkowy, przecinkowy, inkrementacji i dekrementacji

Operator warunkowy

Operator warunkowy ? pozwala na zastąpienie niektórych instrukcji if-else. Operator ten ma postać:

```
wyrażenie1 ? wyrażenie2 :wyrażenie3
```

Operator ten działa w sposób następujący: najpierw jest obliczane wyrażenie1, jeśli ma ono wartość logiczną prawda, wtedy jest obliczane wyrażenie2 i staje się ono wartością całego wyrażenia, jeśli zaś wyrażenie1 ma wartość logiczną fałsz, obliczane jest wyrażenie3 i ono staje się wartością całego wyrażenia.

Przykład 38. *Użycie operatora warunkowego.*

```
1. int x, y;
2. x=5;
3. y=x>10? 50 : 100;
```

Wynikiem obliczenia wyrażenia w linii 3 jest 100, gdyż wyrażenie relacyjne x>100 dla x=5 ma wartość logiczną fałsz.

Przykład 39. *Użycie operatora warunkowego do wyznaczenia większej z dwóch wartości.*

```
z=a>b? a: b; // max(a,b)
```

Zmienna z przybiera wartość równą a, jeśli a>b, albo też b, gdy a ≤ b.

Operator warunkowy (wyrażenie z operatorem warunkowym) może być stosowany wszędzie tam, gdzie w języku C może wystąpić wyrażenie. Priorytet operatora ? jest względnie niski (tuż nad operatorem przypisania), stąd nie jest konieczne stosowanie nawiasów, aby wymusić odpowiednią kolejność wykonywania operacji.

Operator przecinkowy

Użycie operatora przecinkowego można przedstawić w sposób następujący:

```
wyr1, wyr2,…,wyrN
```

Wyrażenia oddzielone przecinkami oblicza się od lewej do prawej, przy czym typem i wartością całego wyrażenia są typ i wartość wyrażenia wyrN.

Przykład 40. *Użycie operatora przecinkowego.*

```
1. int x, y;
2. x= (y=15;y+1);
```

Wartość x jest równa 16.Wynika to stąd, że poszczególne instrukcja są wykonywane od lewej do

prawej. Instrukcja podstawienia y=15 powoduje, że y przybiera wartość 15 i wartość ta jest wykorzystywana przy obliczaniu wyrażenia y+1, które będzie miało wartość 16 i ta wartość staje się wartością całego wyrażenia z operatorem warunkowym.

Operatory inkrementacji i dekrementacji

Dla operatorów tych stosowane są następujące symbole:

++ - dla operatora inkrementacji (zwiększania),

oraz

-- - dla operatora dekrementacji (zmniejszania).

Operatory te zwiększają wartość swego argumentu o 1 (operator ++), lub też zmniejszają o 1 (operator --). Operatory te mogą być używane w następujących formach:

++ operand - operator preinkrementacji,
operand**++** - operator postinkrementacji,
-- operand - operator predekrementacji,
operand **--** - operator postdekrementacji,

Operand w powyższych wyrażeniach z operatorami ++ lub **–** , może być typu arytmetycznego lub wskaźnikowego, musi być jednak tzw. *L*wartością (ang. *Lvalue*). *L*wartość jest to wyrażenie mogące reprezentować pewną lokację pamięci, w najprostszym przypadku jest to zmienna lub też element tablicy. Użycie powyższych operatorów powoduje dwa efekty:

- wartość operandu jest modyfikowana (zwiększana lub zmniejszana o 1),
- wyrażeniu z operandem jest przypisywana pewna wartość.

Działanie operatora postinkrementacji *(*operand **++*)*.

- wartość operandu jest zwiększana o 1,
- wartość wyrażenia operand **++** jest równa początkowej wartości operandu.

Przykład 41. *Operator postinkrementacji*.

```
1. x=3;
2. y=x++;
```

Wynik: y=3, x=4.

Działanie operatora preinkrementacji *(***++**operand*).*

- wartość operandu jest zwiększana o 1,
- wartość wyrażenia ++*operand* jest równa zwiększonej o 1 wartości operandu.

Przykład 42. Operator postinkrementacji.

```
1. x=3;
```

```
2. y=++x;
```
Wynik: y=4, x=4.

*Działanie operatora postdekrementacji (*operand **--).**

- wartość operandu jest zmniejszana o 1,
- wartość wyrażenia operand **--** jest równa początkowej wartości operandu.

Przykład 43. *Operator postdekrementacji*.

```
1. x=3;
2. y=x--;
```
Wynik: y=3, x=2.

Działanie operatora predekrementacji ***(- -*** operand***).***

- wartość operandu jest zmniejszana o 1,
- wartość wyrażenia operand **--** jest zmniejszonej o 1 wartości operandu.

Przykład 44. *Operator predekrementacji*.

```
x=3;
y=--x;
```
Wynik: y=2, x=2.

Przykład 45. *Użycie operatorów preinkrementacji i postinkrementacji*.

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main(int argc, char* argv[])
4. {
5. int x1, x2, y;
6. x1= (y=15,y=y+1,y=y+1,++y);// obliczanie są wyrażenia w nawiasie, począwszy
7.                          // od y=15, wartością całego wyrażenia jest wartość ++y
8. printf("\n x1=%d",x1);
9. x2= (y=15,y=y+1,y=y+1,y++);
10. printf("\n x2=%d",x2);
11. getch();
12. return 0;
13. }
```
Wynik działania programu:
```
x1=18
x2=17
```

9 Tablice

Tablice w języku C są pewnymi strukturami danych stanowiącymi zbiory zmiennych tego samego typu, do których odwołujemy się przy użyciu wspólnej nazwy. Tablice jednowymiarowe mają strukturę w pewnym sensie zbliżoną do struktur matematycznych takich jak wektory, a tablice dwuwymiarowe do macierzy. W języku C tablice umieszczane są w ciągłych obszarach pamięci. Oznacza to umieszczanie reprezentacji elementów tablicy pod kolejnymi adresami pamięci. Najniższy adres odpowiada pierwszemu elementowi tablicy, a najwyższy ostatniemu. Takie rozmieszczenie w pamięci ułatwia operacje na tablicach, gdyż w łatwy sposób można wyznaczać adresy poszczególnych elementów tablicy, znając początkowy adres obszaru pamięci, w którym znajduje się tablica.

Tablice jednowymiarowe

Definicja tablicy jednowymiarowej ma następującą postać:

```
typ nazwa_tablicy [rozmiar];
```

typ określa typ każdego elementu tablicy, rozmiar musi być określony w standardzie C89 przy użyciu wyrażenia stałego, natomiast w standardzie C99 może być zmienną. Dostęp do elementów tablicy uzyskuje się poprzez tzw. indeksowanie. Indeksowanie realizuje się umieszczając indeks (numer elementu tablicy) w nawiasach kwadratowych za nazwą tablicy. W języku C elementy indeksowane są od zera.

Przykład 46. *Indeksowanie*.

```
1. int a[20];
2. a[0]=1; // wpisanie do pierwszego elementu tablicy wartości 1, po lewej stronie zastosowano
3.         //operator indeksowania i  indeks 0
4. a[3]=2;
5. a[5]=2*a[0]-5*a[3]+10; // do elementu a[5] wstawiana jest wartość 2.
```

Elementy tablicy mogą być traktowane w ten sam sposób jak zmienne danego typu.

Poza zastosowaniem instrukcji podstawienia, elementom tablicy można nadać wartości przy użyciu tzw. inicjalizacji lub instrukcji wejścia.

Przykład 47. *Inicjalizacja tablicy.*

```
int a[5]={ 1, 3, -1, 4, 5};
```

Ilość elementów inicjalizujących może być równa lub mniejsza od ilości elementów tablicy. Jeżeli nie stosuje się inicjalizacji przy definicji tablicy, jej elementy mogą mieć wartości przypadkowe, gdy tablica jest definiowana wewnątrz funkcji. Tablice definiowane poza funkcjami bez inicjalizacji (tablice zewnętrzne) są zerowane.

Przykład 48. *Wczytywanie i drukowanie tablicy.*

```
1. #include<stdio.h>
2. #include<conio.h>
3. int main()
4. {
5. double x[5]; // definicja tablicy o 5 elementach typu double
6. int i;
7. for (i=0;i<5;i++)
8. {
9. printf("\n Podaj element x[%d]=",i);
10. scanf("%lf",&x[i]);// wczytywanie elementu tablicy
11. }
12. for (i=0;i<5;i++)
13. printf("\n x[%d]=%f",i, x[i]);// drukowanie elementu tablicy
14. getch();
15. return 0;
16. }
```

Przykładowy wynik działania programu:

```
Podaj element x[0]=1
Podaj element x[1]=20.5
Podaj element x[2]=2
Podaj element x[3]=-5
Podaj element x[4]=2
x[0]=1.000000
x[1]=20.500000
x[2]=2.000000
x[3]=-5.000000
x[4]=2.000000
```

Przykład 49. *Wczytywanie i drukowanie tablicy ze sprawdzaniem formatu.*

```
1. #include<stdio.h>
2. #include<conio.h>
3. int main()
4. {
5. double x[5]; int i;
6. for (i=0;i<5;i++)
7. {
8. do
9. {
10. printf("\n Podaj element x[%d]=",i);
11. k=scanf("%lf",&x[i]);
12. fflush(stdin);
```

```
13. }
14.  while (k==0);
15. }
16.
17. for (i=0;i<5;i++)
18. printf("\n  x[%d]=%f",i, x[i]);
19. getch();
20. return 0;
21. }
```

Komentarz do

Przykład 50. *Sumowanie różnych kategorii elementów tablicy.*

```
1. int a[10]={ 1, 3, -1, 4, 5,6,9,2,15,12};
2. int i, suma=0, suma_p=0, suma_ind=0, ind1=2, ind2=6;
3. for (i=0;i<10;i++)      suma = suma + a[i];
4. for (i=0;i<10;i++)    if (a[i] %2==0)  suma_p = suma_p+a[i];
5. for (i=ind1;i<=ind2 ;i++)      suma_ind = suma_ind + a[i];
```

Przykład 51. *Wyszukiwanie w tablicy elementu minimalnego i maksymalnego.*

```
1. int a[10]={ 1, 3, -1, 4, 5,6,9,2,15,12};
2. int i, amin,  amax;
3. amin=a[0];
4. for (i=1;i<10;i++)
5. if ( a[i] < amin) amin = a[i];
6. amax=a[0];
7. for (i=1;i<10;i++)
8. if ( a[i] > amax) amax= a[i];
9. printf("\n element minimalny =%d ",amin);
10.  printf("\n element maksymalny =%d ",amax);
11.
```

Przykład 52. *Wyszukiwanie w tablicy zadanej wartości i zapis wartości indeksów elementów równych tej wartości.*

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main(int argc, char* argv[])
4. {
5. int a[10]={ 1, 2, -1, 4, 5,6,9,2,15,12};
6. int i, k=0,element, poz[10];
7. printf("\n  Podaj poszukiwany element :");
8. scanf("%d",&element);
9. for (i=0;i<10;i++)
10.  if ( a[i] ==element ) poz[k++]=i;
```

```
11. if (k)
12. {
13. printf("\n Pozycje tablicy na ktorych wystepuje element=%d", element);
14. for (i=0;i<k;i++)
15. printf("   %d    ", poz[i]);
16. }
17. else printf("\n Zadany element nie wystepuje w tablicy");
18. getch();
19. return 0;
20. }
21.
```

10 Funkcje

Paradygmaty programowania

Obecnie istnieją zasadniczo dwa podejścia do tworzenia oprogramowania:
- paradygmat proceduralny (podejście proceduralne)
- paradygmat obiektowy (podejście obiektowe)

Paradygmat proceduralny to wzorzec postępowania, zgodnie z którym program tworzy się z oddzielnie definiowanych podprogramów (funkcji lub procedur), które działają na oddzielnie zdefiniowanych danych. Uważa się, że stosując paradygmat proceduralny można pisać programy obejmujące do 100000 wierszy kodu źródłowego (ograniczenie to jest związane głównie z problemem utrzymania kodu).

Ogólny schemat programu przy podejściu proceduralnym

a) dyrektywy kompilatora
b) prototypy funkcji (deklaracje funkcji)
c) definicje (deklaracje) zmiennych globalnych
d) funkcja main () zawierająca definicje zmiennych lokalnych, instrukcje i wywołania funkcji
e) definicje funkcji

Duże programy zgodnie z paradygmatem proceduralnym można dzielić na funkcje i towarzyszące im, ale oddzielne struktury danych.

Komunikacja między poszczególnymi funkcjami odbywa się poprzez przekazywanie pewnych wartości do funkcji, zwracanie wartości jak również poprzez przekazywanie adresów pewnych zmiennych do funkcji, które to funkcje mogą te zmienne modyfikować.

Ogólna budowa programu w języku C

W języku C program składa się z jednego lub większej liczby segmentów kodu zwanych funkcjami, z których jedna musi nazywać się main().Wykonanie programu zaczyna się od funkcji main() i program powinien kończyć się wraz z zakończeniem działania funkcji main(). Program również może się zakończyć, gdy pojawi się sytuacja nienormalna-wyjątek, np. dzielenie przez zero, albo też gdy zostanie zakończony w sposób jawny (ctrl-break, zamknięcie okna w systemie Windows), powodem zakończenia może być też wywołanie funkcji exit().

Powody stosowania funkcji w programach

1. Korzystanie z bibliotek lub funkcji stworzonych przez innych programistów.
2. Podział zadania programistycznego na mniejsze, które mogą być opracowywane niezależnie(np. w różnym czasie lub równocześnie przez różnych programistów).
3. Ułatwienia w uruchamianiu dużych programów (można niezależnie uruchamiać i testować poszczególne funkcje).
4. Program jest zbyt duży, by pisać go jako jedną funkcję main().
5. Unikanie powtarzania w programie tego samego kodu, gdy pewne zadania trzeba wykonywać wielokrotnie.

Funkcje a struktura programu w języku C

Program w języku C może być programem jednoplikowym lub wieloplikowym. W programie jednoplikowym wszystkie funkcje znajdują się w jednym pliku źródłowym lub są do niego dołączane przy zastosowaniu dyrektywy #include. W programie wieloplikowym pliki są łączone w sposób zależny od środowiska (np. w środowisku Dev C++ opcja *Projekt->Dodaj do projektu)*.

-Funkcje nie muszą występować w pliku źródłowym w określonym porządku (pewne sposoby uporządkowania mogą wymagać dodatkowych deklaracji), jednak umieszczenie main() jako pierwszej funkcji w pliku ułatwia analizę programu.

-Każda funkcja może wywoływać dowolną inną funkcję.

-Nie jest dopuszczalne zagnieżdżanie funkcji, tzn. nie można jednej funkcji definiować wewnątrz innej funkcji.

-W programie występuje tylko jedna funkcja main() i musi ona wystąpić niezależnie od liczby plików źródłowych.

Przykład 53. *Prosty program stosujący funkcję do obliczania kwadratu pewnej wartości.*

```
1. #include <stdio.h>
2. int kwadrat (int num)
3. {
```

```
4.  int pom; // zmienna lokalna funkcji
5.  pom=num*num;
6.  return pom;
7.  }  // zmienna pom nie jest konieczna, może być  return num*num;
8.
9.  int main()
10.  {
11.  int kw, wartosc;
12.  printf("\n Podaj wartosc :");
13.  scanf("%d",&wartosc);  /* zmienna wartosc jest tzw. argumentem funkcji, zmienna wartość zawiera
     liczbę, która ma być podniesiona do kwadratu *
14.  kw=kwadrat(wartosc);
15.  printf ("\n Kwadrat liczby %d  wynosi :%d",wartosc, kw);
16.  system("PAUSE");
17.  return 0;
18.  }
```

W momencie zakończenia działania, funkcja `kwadrat` przesyła do funkcji `main()` obliczoną wartość i wartość ta jest wstawiana w miejsce wywołania funkcji. W kolejnym etapie następuje wstawienie tej wartości do zmiennej `kw`.

Przykład 54. *Zastosowanie deklaracji funkcji*.

```
1.  #include <stdio.h>
2.  int kwadrat (int); // deklaracja funkcji
3.  int main()
4.  {
5.  int kw, wartosc;
6.  printf("\n Podaj wartosc :"); scanf("%d",&wartosc);
7.  kw=kwadrat(wartosc);
8.  printf("\n Kwadrat liczby %d wynosi:%d \n", wartosc,kw);
9.  system ("PAUSE");
10.  return 0;
11.  }
12.
13.  int kwadrat (int num)
14.  {
15.  int pom;
16.  pom=num*num;
17.  return pom;
18.  }
```

Wynik działania programu

```
----------------------
Podaj wartosc :25
Kwadrat liczby 25 wynosi:625
```

Przykład 55. *Napisać i wywołać w main() następujące funkcje:*
a) funkcję obliczającą sumę swoich argumentów typu int
b) funkcję obliczającą różnicę swoich argumentów typu int
c) funkcję obliczającą iloczyn swoich argumentów typu double

```
1. #include <stdio.h>
2. int sum (int a, int b);  // deklaracje funkcji
3. int roznica (int a, int b);
4. double iloczyn (double a, double b);
5. int main()
6. {
7. int a,b,c,d,sum,roz;
8. double e,f,ilo;
9. printf("\n Podaj a :");
10. scanf("%d",&a);
11. printf("\n Podaj b :");
12. scanf("%d",&b);
13. sum=suma(a,b);// a,b to argumenty wywołania funkcji printf("\n Suma %d + %d =%d ", a,b,sum);
14. printf("\n Podaj c :");
15. scanf("%d",&c);
16. printf("\n Podaj d :");
17. scanf("%d",&d);
18. sum=suma(c,d);
19. printf("\n Suma %d + %d =%d ", c,d,sum);
20. roz=roznica(a,b);
21. printf("\n Roznica %d - %d =%d ", a,b,roz);
22. roz=roznica(c,d);
23. printf("\n Roznica %d - %d =%d  \n", c,d,roz);
24. printf("\n Podaj e :");
25. scanf("%lf",&e);
26. printf("\n Podaj f :");
27. scanf("%lf",&f);
28. ilo=iloczyn(e,f);
29. printf("\n Iloczyn %f * %f =%f \n", e,f,ilo);
30. system ("PAUSE");
31. return 0;
32. }
33. int suma (int a, int b)
34. {
35.  int sum;
36.  sum=a+b;
37.  return sum;
38. }
39.
40. int roznica (int a, int b)
```

```
41. {
42.  int roz;
43.  roz=a-b;
44.  return roz;
45. }
46.
47. double iloczyn (double a, double b)
48. {
49. double ilo;
50. ilo=a*b;
51. return ilo;
52. }
```

Wynik działania programu

```
--------------------------------
 Podaj a :2
 Podaj b :3
 Suma 2 + 3 =5
 Podaj c :4
 Podaj d :5
 Suma 4 + 5 =9
 Roznica 2 - 3 =-1
 Roznica 4 - 5 =-1
 Podaj e :2.5
 Podaj f :3.5
 Iloczyn 2.500000 * 3.500000 =8.750000
```

Przykład 56. Napisać i zastosować w programie funkcję silnia.

```
1. #include <stdio.h>
2. int silnia (int n )
3. {
4. int i, sil=1;
5. if (n>0)
6. for (i=1;i<=n;i++)
7. sil=sil*i;
8. return sil;
9. }
10.  int main()
11. {
12. int l1=5, l2=6;
13. int sil1,sil2;
14. sil1=silnia (l1);
15. sil2=silnia (l2);
16. printf("\n % d ! wynosi %d", l1,sil1);
17. printf("\n % d ! wynosi %d\n", l2,sil2);
18. system("PAUSE");
19. return 0;
```

```
20. }
```

Wynik działania programu

```
 5 ! wynosi 120
 6 ! wynosi 720
```

Przykład 57. *Napisać i zastosować w programie funkcję* wczytajd *wczytującą zmienną typu* double.

```
1.  #include <stdio.h>
2.  double wczytajd  (void )
3.  {
4.  int k; double x;
5.  do
6.  {
7.  printf("\n Podaj zmienna :");
8.  k=scanf("%lf",&x);
9.  if (k==0) printf("\n Niepoprawny format liczby !");
10.  fflush(stdin);
11.  }
12.  while (k==0); // można też   while (!k);
13.  return x;
14.  }
15.
16.  int main()
17.  {
18.  double x1,x2;
19.  x1=wczytajd();
20.  x2=wczytajd();
21.  printf("\n x1 wynosi %f", x1);
22.  printf("\n x2 wynosi %f \n", x2);
23.  system("PAUSE");
24.  return 0;
25.  }
```

Wynik działania programu

```
------------------------
Podaj zmienna :12
Podaj zmienna :23
x1 wynosi 12.000000
x2 wynosi 23.000000
```

Przykład 58 . Użycie funkcji silnia w obliczania wartości częściowej sumy szeregu szeregu.

```
1.  #include <stdio.h>
2.  #include <math.h>// włączenie pliku nagłówkowego z bibliotecznymi funkcjami
3.                          // matematycznymi
4.  double silnia (int);
```

```
5.  int main(int argc, char *argv[])
6.  {
7.  int i,j,n;
8.  double xn,s,sil,x;
9.   while(1)// wielokrotne wykonanie programu
10.  {
11.  printf("\n Podaj x:");
12.  scanf("%lf",&x);
13.  printf("\n Podaj liczbe wyrazow szeregu:");
14.  scanf("%d",&n);
15.  s=0;
16.  for ( i=0;i<n;i++)
17.  {
18.  xn=pow(x,i)/silnia(i);// funkcja pow ( x,i) realizuje potęgowanie
19.  s=s+xn;
20.  }
21.  printf("\n Wartosc e_do_x z szeregu:%f",s);
22.  printf("\n Wartosc e do x: exp      %f\n", exp(x));
23.  }// koniec  while
24.  system("pause");
25.  return 0; }
26.
27.  double silnia (int n)
28.  {
29.  int i; double sil=1;
30.  for (i=1;i<=n;i++)
31.  sil=sil*i;
32.  return sil;
33.  }
```

Wynik działania programu

```
----------------------Podaj x: 2
Podaj liczbe wyrazow szeregu:6
Wartosc e_do_x z szeregu:7.266667
Wartosc e do x: exp       7.389056
Podaj x: 2
Podaj liczbe wyrazow szeregu: 7
Wartosc e_do_x z szeregu:7.355556
Wartosc e do x: exp       7.389056
Podaj x: 2
Podaj liczbe wyrazow szeregu:8
Wartosc e_do_x z szeregu:7.380952
Wartosc e do x: exp       7.389056
}
```

Ogólna budowa funkcji

Definicja funkcji

typZwracanejWartości nazwaFunkcji (deklaracjaParametrówFormalnych, jeśli występują)
```
{
        deklaracje
        instrukcje
}
```

Definicje poszczególnych funkcji mogą się pojawić w jednym pliku lub większej liczbie plików źródłowych, przy czym żadna funkcja nie może być rozdzielona między plikami źródłowymi (definicja funkcji musi być umieszczona w jednym pliku).

Deklaracja funkcji

Deklaracja funkcji stanowi informację dla innych funkcji stosujących daną funkcję o typie zwracanej wartości, liczbie i typach parametrów funkcji. Deklaracja ma postać skróconego nagłówka funkcji zakończonego średnikiem. Typ void jako typ zwracany przez funkcję oznacza, że funkcja nie zwraca żadnej wartości, jeżeli występuje zamiast listy parametrów formalnych, oznacza,że funkcja nie ma parametrów.

Przykład 59. *Deklaracje funkcji*.

```
1. int kwadrat(int, int);
2. int echoLine(void); // void oznacza tutaj, że funkcja nie ma
3.                     //parametrów
4. void printHist(int size);// void oznacza tutaj, że funkcja nic nie
5.                          //zwraca
6. void printPrompt(void);
```

Nazwy po typach parametrów są opcjonalne i są ignorowane przez kompilator. Nazwy parametrów stosować można dla celów dokumentacji, pozwalają one zorientować się co do roli poszczególnych parametrów bez sięgania do definicji funkcji.

Definicja a deklaracja funkcji

Definicja funkcji może występować jednokrotnie.

1. Deklaracja funkcji f może występować wewnątrz funkcji g, która wywołuje daną funkcję f lub też na zewnątrz wszystkich funkcji.

2. Deklaracja funkcji f wewnątrz funkcji g służy tylko funkcji g.
3. Deklaracja funkcji f na zewnątrz wszystkich funkcji służy wszystkim tym funkcjom, których definicje umieszczono po deklaracji funkcji f .

Deklaracja funkcji wywoływanej wewnątrz funkcji wywołującej	Deklaracja funkcji wywoływanej przed funkcją wywołującą

// funkcja g - definicja
{
// deklaracja funkcji f
 	
f(...); // wywołanie f
}
	// deklaracja funkcji f

// funkcja h -definicja
{
f(...); // wywołanie f
}
	// funkcja h -definicja
	{
	f(...); // wywołanie f
	}

// funkcja f -definicja
{
 	
}
	// funkcja f -definicja
	{
	
	}

Instrukcja return

Instrukcja return (ang. powróć, zwróć) pełni dwie zasadnicze funkcje:

-zwraca sterowanie programem do miejsca wywołania funkcji (do funkcji wywołującej).
-umożliwia przekazanie wartości obliczonej przez funkcję do funkcji wywołującej.

Funkcja, która zwraca wartość, powinna mieć przynajmniej jedną instrukcję return w formie

```
return (wyrażenie);
```

lub

```
return wyrażenie;
```

gdzie wyrażenie jest jakimkolwiek legalnym wyrażeniem języka C.

- obie formy mają to samo znaczenie.
- do funkcji wywołującej zwracana (przekazywana) jest wartość wyrażenia.
- funkcja przy wywołaniu może zwrócić co najwyżej jedną wartość.

Przykład 60. Instrukcja return.

```
a) return  25*5 + fun2(x);
b) return (25*5+ fun2(x));
```

```
c) double modul ( double x)
{
return x>0 ? x : -x;
}
```

Niezależnie od tego czy wyrażenie po return jest skomplikowane, czy też nie, zwracana jest jedna wartość. Funkcja może_posiadać kilka instrukcji return, jednak w czasie danego wywołania funkcji może zostać wykonana tylko jedna instrukcja return. Nie powinno się stosować w funkcji kilku instrukcji return, gdyż funkcja staje się mniej zrozumiała oraz trudniejsza do testowania i modyfikacji. Instrukcja return nie jest konieczna w funkcji, która nie zwraca żadnej wartości, ale jeśli jest używana to w postaci return. Jeśli nie ma instrukcji return, wykonywane są kolejne instrukcje w treści funkcji, aż do momentu dojścia do nawiasu zamykającego funkcję. Przekazywanie z funkcji więcej niż jednej wartości będzie przedstawiome w p.1.12.

11 Wskaźniki i dynamiczna alokacja pamięci

Wskaźniki to typ obiektów, przechowujących adresy innych obiektów np. zmiennych. Adres to konkretna lokalizacja pewnego obiektu w pamięci. Jeżeli pewien obiekt typu wskaźnikowego (np. zmienna) przechowuje adres innego obiektu, to mówimy, że wskazuje na ten obiekt.

```
int main()
{
int x=5 ,y=23;
int *px;// definicja zmiennej wskaźnikowej
px=&x;
}
```

12F88	5	23
px	y	x

12FF80	12FF84	12FF88	adresy hex
(1245056)	(1245060)	(1245064)	adresy dec

Adresy te mogą się zmieniać przy kolejnych uruchomieniach programu, jednak w komórce px będzie zawsze adres zmiennej x.

Przykład 61. *Program obliczający adresy zmiennych i zawartość zmiennej wskaźnikowej.*

```
1. #include <stdio.h>
2. int main(int argc, char *argv[]) {
3. int x,y;
4. int *px;
```

```
5. printf("\n Adres zmiennej x  w hex   = %p  i w  dec =%d\n", &x,&x);
6. printf("\n Adres zmiennej y   w hex   = %p  i w  dec =%d\n", &y,&y);
7. printf("\n Adres wskaznika px  w hex   =%p  i w  dec =%d\n",&px,&px);
8. px=&x;
9. printf("\n Zawartosc komorki px w hex  =%x   i w  dec =%d\n", px,px);
10. system("PAUSE");
11. return 0;
12. }
```

Przykładowy wynik działania programu:

```
Adres zmiennej x  w hex      = 0012FF88  i w  dec =1245064
Adres zmiennej y  w hex      = 0012FF84  i w  dec =1245060
Adres wskaznika px w hex     =0012FF80  i w  dec =1245056
Zawartosc komorki px w hex  =12ff88      i w  dec =1245064
```

Definicja prostej zmiennej wskaźnikowej

Definicja zmiennej wskaźnikowej składa się z nazwy typu wskazywanych obiektów, gwiazdki oraz nazwy zmiennej

```
typ_wskazywanych_obiektów   * nazwa_zmiennej ;
```

Typ_wskazywanych_obiektów zwany jest zwykle typem wskaźnika.

Przykład 62. *Definicja zmiennej wskazującej na obiekty typu* int.

```
int * px;
```

Operatory wskaźnikowe

- operator adresowy &

- operator adresowania pośredniego *(dereferencji)

Jednoargumentowy operator & przyłożony do operandu zwraca adres operandu w pamięci (dokładniej - wyrażenie `&operand` reprezentuje adres operandu).

Jednoargumentowy operator * przyłożony do operandu wskaźnikowego, zwraca wartość znajdującą się pod adresem zawartym w obiekcie (zmiennej) występującym po operatorze.

Przykład 63. *Proste działania przy użyciu wskaźników.*

```
1.  int *px, *py;  // definicja zmiennej wskaźnikowej px
2. int i=5, j,k=10;
3. px=&i;// przypisanie adresu zmiennej i wskaźnikowi px;
4. j=*px;// podstawienie i do j  przy użyciu wskaźnika
5. *px= i+k; // podstawienie sumy i+k do i
6. py=px;// przypisanie wskaźnikowi py wskaźnika px, obydwa wskaźniki wskazują
7. na ten sam obszar pamięci
```

```
8.  *py=20; // podstawienie do i wartości 20
9.  printf(”\n i=%d”, *py); // drukowanie i przy użyciu
10.                    //wskaźnika py
11.  *py=*py+5;
12.  printf(”\n i=%d”, *py);
```

Wyrażenie *p ma różny sens w zależności od tego czy występuje po lewej czy też po prawej stronie instrukcji podstawienia. Dla następujących definicji i instrukcji

```
1.  int x=5,y, *px;
2.  px=&x;
```

w instrukcji

```
y=*px;
```

wyrażenie *px reprezentuje zawartość komórki wskazywanej przez px, czyli komórki x (jest to 5). Natomiast w instrukcji

```
*px=15;
```

wyrażenie *px reprezentuje adres pamięci (zmienną), pod który można wpisać jakąś wartość.

Przykład 64. *Wczytywanie i drukowanie zmiennej przy użyciu wskaźnika*.

```
1.  #include <stdio.h>
2.   int main()
3.  {
4.  double *p;
5.  double x;
6.  p=&x;// przypisanie wskaźnikowi p adresu zmiennej x
7.  printf("\n Podaj x:");
8.  scanf("%lf", p);// wczytanie zmiennej x przy użyciu wskaźnika
9.  printf("\n x=%f",*p);// wydruk zmiennej x za pośrednictwem wskaźnika
10.  system("PAUSE");
11.  return 0;
12.  }
```

Arytmetyka na wskaźnikach

W języku C stosując wskaźniki można posługiwać się tzw. ograniczoną arytmetyką na wskaźnikach. Oznacza to, że można stosować niektóre operacje arytmetyczne, by obliczyć odpowiedni wskaźnik. Należy jednak w każdym przypadku zważać na to, co może oznaczać wynik danego działania. Dla wskaźników można stosować następujące operacje wskaźnikowe:

Działania arytmetyczne na wskaźnikach

Na wskaźnikach wskazujących na ten sam typ można dokonywać następujących operacji:

-inkrementacji (++) (zwiększania)
-dekrementacji (--) (zmniejszania)
-dodawania do wskaźnika liczby całkowitej
-odejmowania od wskaźnika liczby całkowitej
-odejmowania wskaźników od siebie
-porównywania dwóch wskaźników.

Stosowany poniżej operator sizeof, użyty dla typu zwraca rozmiar typu w bajtach.

Przykład 65. Dla celów ilustracji przedstawionych w dalszym ciągu działań w arytmetyce wskaźników przyjmijmy

```
1. int a[10];
2. int *wsk;
3. wsk=a; .
```

Inkrementacja wskaźnika

Jeżeli wskaźnik wsk do pewnego typu wskazuje na pewną komórkę pamięci to wyrażenie wsk++ wskazuje na komórkę o adresie wsk+sizeof(typ), czyli wsk+4, a więc na a[1].

Dekrementacja wskaźnika

Jeżeli wskaźnik wsk do pewnego typu wskazuje na pewną komórkę pamięci to wyrażenie wsk-- wskazuje na komórkę o adresie wsk-sizeof(typ), czyli wsk-4, zakładając,że wskaźnik wskazywał na a[1], po dekrementacji będzie wskazywał na a[0].

Dodawanie do wskaźnika liczby całkowitej

Jeżeli wskaźnik wsk do pewnego typu wskazuje na pewną komórkę pamięci to wyrażenie wsk+liczba wskazuje na komórkę o adresie

wsk+liczba*sizeof(typ)

a więc działanie wsk+3 spowoduje (przy założeniu, że przed dodaniem 3 wskazywał na a[0]), że wskaźnik będzie wskazywał na a[3].

Odejmowanie od wskaźnika liczby całkowitej

Jeżeli wskaźnik wsk do pewnego typu wskazuje na pewną komórkę pamięci, to wyrażenie wsk-liczba wskazuje na komórkę o adresie

```
wsk-liczba*sizeof(Typ).
```

Jeśli wskaźnik wsk wskazuje na a[3], to po wykonaniu działania wsk-2, będzie wskazywał na komórkę a[1], gdyż zostanie cofnięty o 8 bajtów.

Odejmowanie wskaźników

Jeżeli dane są dwa wskaźniki wsk1 i wsk2 do tego samego typu Typ, to po wykonaniu instrukcji

```
diff= wsk2 - wsk1;
```

to diff ma wartość dodatnią, jeżeli wsk2 wskazuje na wyższy adres i ma wartość równą liczbie elementów typu Typ, które można umieścić między adresami komórek pamięci wskazywanych przez wsk1 i wsk2. Jeśli wsk2 wskazuje na adres niższy, diff ma wartość ujemną, a interpretacja jest podobna. Jeśli mamy

```
1. int a[5];
2. wsk1=a;
3. wsk2=&a[2];
```

to wynikiem działania wsk2-wsk1 będzie 2.

Porównywanie wskaźników

Jeśli wskaźniki wsk1 i wsk2 wskazują na ten sam typ, to wyrażenie

```
wsk2>wsk1
```

ma wartość logiczną prawda, jeżeli wsk2 wskazuje na adres wyższy i wartość fałsz, gdy na niższy. Jeśli przykładowo, wsk2=&a[3], a wsk1=a[0], to wsk2>wsk1, będzie miało wartość prawda.

Wskaźniki i tablice

Nazwa tablicy w C jest stałym wskaźnikiem do pierwszego elementu tablicy. Np. jeżeli stworzymy definicje

```
int a[5];
int *pa;
```

i zastosujemy instrukcję

```
pa=a;
```

to wskaźnik pa będzie wskazywał na a[0], czyli pierwszy element tablicy a. Stosując wskaźniki można wykonać te same operacje na tablicach, co przy użyciu indeksowania.

Przykład 66. *Proste działania na tablicach przy użyciu wskaźników.*

```
1. #include <stdio.h>
2. int main()
3. {
4.  int a[5];int *pa,i;
5. pa=a;// przypisanie wskaźnikowi pa adresu a[0]
6. *pa=10; //zapis wartości 10 do a[0]
7. pa++; //zwiększenie wskaźnika, wskazuje on teraz na a[1]
8. *pa=20; // zapis wartości 20 do a[1];
9. pa=pa+1; // wskaźnik wskazuje na element a[2]
10. for (i=2; i<5; i++){
11. printf("\n Podaj a[%d]=",i);
12. scanf("%d",pa);
13. pa++;
14. }
15. pa=a; // ustawienie wskaźnika na początek tablicy
16.       // alternatywnie pa= pa-5
17. for (i=0;i<5;i++)
18. printf("\n a[%d]=%d", i,*pa++);
19. printf("\n");
20. system("PAUSE");
21. }
```

Język C – dynamiczna alokacja pamięci

W języku C istnieją zasadniczo dwa rodzaje zmiennych:

-zmienne zwykłe definiowane w funkcji main() lub na zewnątrz wszystkich funkcji lub w pewnym bloku. Każda zmienna tego rodzaju posiada swoją nazwę oraz określony typ.

- zmienne dynamiczne tworzone i usuwane w trakcie działania programu; taki sposób przydzielania pamięci zwany jest alokacją w trakcie działania programu (ang. run-time allocation). Zmienne te nie posiadają nazw, znane są wyłącznie adresy przydzielonej im pamięci (wskaźniki do tej pamięci). Do przydzielania pamięci zmiennym dynamicznym służą w języku C funkcje malloc i calloc. Do usuwania zmiennych dynamicznych stosuje się funkcję free.

Funkcje malloc i calloc (stdlib.h)

Każda z tych funkcji alokuje przydziela pamięć i zwraca adres tej pamięci (wskaźnik do tej pamięci). Rozmiar przydzielanej pamięci nie musi być znany podczas kompilacji.

Funkcja malloc

Nagłówek funkcji tej ma postać następującą:

```
void * malloc (int);
```

Funkcja malloc oczekuje, jako swojego argumentu, liczby bajtów, które mają być przydzielone w danym wywołaniu funkcji. Jeżeli przydzielenie pamięci jest możliwe, funkcja zwraca wskaźnik do tej pamięci, jeśli nie, funkcja zwraca `NULL` (zerowy wskaźnik). Zwracany wskaźnik jest typu void*, czyli jest to wskaźnik do void. Wskaźnik ten musi być przekształcony (przez rzutowanie) na wskaźnik do żądanego typu. Język C gwarantuje, że wskaźnik do `void` może być przekształcony na wskaźnik do każdego innego typu.

Przykład 67. *Zastosowanie funkcji* `malloc` *do alokacji pamięci dla zmiennej dynamicznej typu* `int`.

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.  int *ptr;
5.  ptr=(int*) malloc( sizeof(int));//wywołanie funkcji malloc alokującej blok
6.  rozmiarze 4 bajtów, jest to wartość określona przez operator
7.  sizeof dla typu int. Otrzymany wskaźnik typu void jest rzutowany na typ int
8.  //( działanie (int*). Celem alokacji jest uzyskanie miejsca pamięci dla zmiennej
9.  //dynamicznej typu int
10.  if (ptr= = NULL) // W sytuacji, gdy alokacja nie może być zrealizowana, funkcja
11. // malloc zwraca zerowy wskaźnik
12.  {
13.  printf("\n Przydzielenie pamięci nie było możliwe");
14.  system("pause");
15.  return 1;
16.  }
17.  printf(" Podaj wartosc :");
18.  scanf("%d", ptr);// wczytanie zniennej dynamicznej typu int
19.  printf("\n Wartosc to :", *ptr);// wydruk zmiennej dynamicznej
20.  free(ptr);// dealokacja pamięci
21.  system("pause");
22.  return 0;
23.  }
```

Funkcja calloc

Prototyp funkcji tej ma postać następującą:

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

Funkcja calloc oczekuje dwóch argumentów typu int. Pierwszy argument oznacza liczbę bloków pamięci, które mają zostać przydzielone, a drugi rozmiar pojedynczego bloku. Funkcja zwraca wskaźnik do pierwszego bloku. Wskaźnik ten jest typu void* i musi być rzutowany na wskaźnik do wymaganego typu.

Przykład 68. *Zastosowanie funkcji* calloc *do alokacji pamięci dla tablicy liczb typu* int.

```
1. #include <stdio.h>
2. int main()
3. {
4.  int *ptr ;// definicja zmiennej ptr jako wskaźnika do int
5.  int i;
6. ptr=(int ) calloc(5, sizeof(int));/ / wywołanie funkcji calloc alokującej 5 bloków
7.  pamięci, każdy o rozmiarze 4 bajtów, jest to wartość określona przez operator
8.  sizeof dla typu int. Otrzymany wskaźnik typu void jest rzutowany na typ int
9.  //( działanie (int*). Celem alokacji jest uzyskanie miejsca pamięci dla tablicy dynamicznej
10. //  typu int o 5 elementach
11.
12.  for (i=0;i<5;i++) // wczytywanie tablicy dynamicznej przy użyciu wskaźnika
13.  {
14.  printf("\n Podaj element %d", i);
15.  scanf("%d", ptr++); // po wczytaniu każdego elementu wskaźnik jest inkrementowany o
16.  ( zwiększany) o  rozmiar typu int ( 4) .
17.  }
18.  ptr=ptr-5;// cofnięcie wskaźnika na początek alokowanego obszaru, aby można
19.  // było odczytać i wydrukować wartości wczytanej tablicy dynamicznej
20.  for (i=0;i<5;i++) { // wydruk tablicy
21.  printf("\n Element [%d]=%d", *ptr++);
22.  ptr-=5;// ptr=ptr-5;,przesunięcie wskaźnka na początek alokowanego obszaru przez jego dealokacją
23.  free(ptr);
24.  system("pause");
25.  return 0;
26.  }
```

12 Komunikacja funkcji z otoczeniem

Każda funkcja w C może dysponować mechanizmami komunikacji z otoczeniem. Funkcja może pobierać wartości lub adresy od funkcji wywołującej, może zwracać wartość do funkcji wywołującej, jak również modyfikować argumenty, których adresy zostały do niej przekazane.

Istnieją dwa główne sposoby przekazywania argumentów:

- przekazywanie przez wartość
- przekazywanie przez zmienną (przekazywanie przez referencję, przez wskaźnik)

Przekazywanie przez wartość(ang. call by value)

Przy przekazywaniu przez wartość, wartości argumentów są kopiowane do parametrów formalnych, zmiany parametrów formalnych nie mają wpływu na wartości argumentów (potocznie mówi się, że funkcja działa na kopiach argumentów). Przy takim sposobie przekazywania parametrów funkcja wywoływana nie może zmienić argumentów zastosowanych przy wywołaniu. Argument może być wyrażeniem, którego wartość jest najpierw obliczana i następnie przekazywana do funkcji wywołującej.

Przykład 69. *Przekazywanie przez wartość*.

```
1. #include <stdio.h>
2. int sqr (int);  // deklaracja funkcji
3. int main()
4. {
5. int t=10, kw;
6. kw=sqr(t); // argument wywołania - zmienna t nie jest modyfikowana, pomimo że  parametr formalny
7.              // odpowiadający  t,  jest modyfikowany.
8. printf ("\n Kwadrat liczby %d  wynosi :%d",t, kw);
9. system("pause");
10. return 0;
11. }
12. int sqr (int k)// definicja funkcji sqr, funkcja otrzymuje jeden argument typu int
13. podnosi do kwadratu i zwraca obliczoną wartość do funkcji main()
14. {
15.  k=k*k;
16.  return k;
17. }
18. // Przypisanie k=k*k modyfikuje tylko lokalną zmienną k.
```

Należy zauważyć, że użycie funkcji do obliczenia kwadratu liczby nie jest potrzebne, natomiast ta prosta operacja posłużyła do ilustracji mechanizmu przekazywania parametru przez wartość.

Przekazywanie przez zmienną

Przy przekazywaniu przez zmienną, do funkcji przekazuje się wskaźnik do argumentu (adres argumentu), a nie jego wartość. Funkcja wywoływana dysponując tym wskaźnikiem może zmienić argument znajdujący się poza funkcją. Mechanizm przekazywania przez zmienną umożliwia przekazanie na zewnątrz funkcji więcej niż jednej wartości.

Przykład 70. *Funkcja zwracająca dwie wartości.*

```
1. #include <stdio.h>
2. int dodajMnoz (int,int,int *);
3. int main()
4. {
5. int x=10, y=20, suma, iloczyn;
6. suma=dodajMnoz(x,y,&iloczyn);// wywołanie funkcji dodajMnoz, wartość sumy
7.  // przekazywana poprzez instrukcję return jest przypisywana do zmiennej suma,
8.  // a wartość iloczynu jest zapisywana do komórki o adresie &iloczyn, czyli do zmiennej
9. //  iloczyn
10.  printf("\n Suma= %d  iloczyn=%d ", suma, iloczyn);
11.   system("pause");
12.  return 0;
13.  }
14.
15.  int dodajMnoz (int a,int b,int* pc)// definicja funkcji obliczającej sumę i iloczyn
16.   // swoich argumentów, ponieważ przy użyciu return możliwe jest przekazanie tylko
17. // jednej wartości, a należy przekazać dwie, do przekazania wartości iloczynu jest
18.   //wykorzystywany wskaźnik pc
19.  {
20.  int sum, ilo;
21.  sum=a+b;
22.  ilo=a*b;
23.  *pc=ilo; // wstawienie w miejsce pamięci wskazywane przez pc, wartości
24. //  iloczynu.  Instrukcja pokazuje realizowane działanie, natomiast rzeczywisty
25.   // adres jest podawany w czasie wywołania funkcji
26.  return sum;
27.  }
```

Przykład 71. *Funkcja zamieniająca swe argumenty.*

```
1. #include<stdio.h>
2. void swap (int*,int*);// deklaracja funkcji
3. int main()
4. {
5. int i=10, j=20;
6. printf("\n Wartości i oraz j przed zamianą i=%d j=%d:",i,j);
7. swap(&i, &j); // wywołanie funkcji zamieniającej swe argumenty,
8. // do funkcji przekaywane są adresy komórek i  i  j.
9. printf("\n Wartości i oraz j po zamianie i=%d j=%d:", i,j);
10.  system ("pause");
11.  return 0;
12.  }
13.  void swap (int* px, int *py)
14.  {
15.  int temp=*px; //zapisz do temp wartość znajdującą się pod  adresem px
```

```
16.  *px=*py;       // umieść w komórce wskazywanej przez px, wartość znajdującą się pod adresem
17.  // wskazywanym przez py
18.  *py=temp; // umieść w komórce wskazywanej przez py, wartość z komórki temp
19.  }
```

Funkcja nie zawiera instrukcji return, gdyż nie zwraca żadnej wartości przy zastosowaniu tej instrukcji (typem zwracanym jest void). Instrukcja return mogłaby wystąpić w formie return; lecz może być pominięta.

Przekazywanie tablic jako argumentów

Gdy argumentem funkcji wywoływanej jest tablica, do funkcji przekazywany jest jej adres (adres pierwszego elementu). W związku z tym funkcja operuje na przekazanej jej tablicy, a nie na kopii. Funkcja ma możliwość zmiany zawartości tablicy.

Przykład 72. *Funkcja oblicza sumę elementów tablicy i umieszcza tę sumę w pierwszym elemencie tablicy.*

```
1.  #include <stdio.h>
2.  void sumaTab(int [ ], int);// deklaracja funkcji sumaTab
3.  int main()
4.  {
5.  int sum, a[5]= {0,1,2,-1,3};// definicja i inicjalizacja tablicy a
6.  sumaTab(a,5); // wywołanie funkcji obliczającej sumę elementów tablicy a i
7.                  // umieszczającą tę sumę w komórce a[0]
8.  printf("\n Suma elementów tablicy a=%d", a[0]);
9.  system("pause");
10.  return 0;
11.  }
12.  void sumaTab(int b[ ],int n)
13.  {
14.  int i,suma=0;
15.  for (i=0;i<n; i++)// obliczanie sumy elementów tablicy
16.  suma=suma+b[i];
17.  b[0]=suma;  // obliczona suma jest umieszczana w pierwszym elemencie tablicy
18.  // przekazanej podczas wywołania funkcji
19.  }
```

Przykład 73. *Funkcje wczytujące i drukujące tablicę jednowymiarową.*

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  void  wczytTab( int [ ], int );
4.  void  drukTab( int [ ], int );
5.  int main()
6.  {
```

```
7.  int a[5],b[10];
8.  wczytTab(a,5);// wywołanie funkcji wczytTab wczytującej 5-elementową tablicę a
9.  wczytTab(b,10);
10.  drukTab(a,5);// wywołanie funkcji wczytTab drukującej 5-elementową tablicę a
11.  drukTab(b,10);
12.  getch();
13.  return 0;
14.  }
15.  void  wczytTab( int x[ ], int n)// definicja funkcji wczytującej tablicę
16.                                    // jednowymiarową  typu int o n elementach
17.   {
18.  int i;
19.  for (i=0;i<n; i++)
20.  {
21.  printf("\n Podaj element %d=",i);
22.  scanf("%d",&x[i]);
23.  }
24.  }
25.
26.  void drukTab( int x[ ], int n)// definicja funkcji drukującej tablicę
27.                                 // jednowymiarową typu int o n elementach
28.  {
29.  int i;
30.  for (i=0;i<n; i++)
31.   printf("\n  Element %d=%d ",i, x[i]);
32.  }
```

Przykład 74. *Funkcja obliczająca w tablicy sumy liczb parzystych, nieparzystych, dodatnich i ujemnych.*

```
1.  #include <stdio.h>
2.  void  wczytTab( int [ ], int ); //deklaracje funkcji wczytTab
3.  int obliczSumy(int [ ], int n, int *pnp, int *pd, int *pu); // deklaracja funkcji
4.                                                              // obliczSumy
5.  int main()
6.  {
7.  int  a[10], sumaP, sumaNp, sumaD, sumaU;
8.  wczytTab(a,10);
9.  sumaP=obliczSumy(a,10,&sumaNp, &sumaD, &sumaU);// wywołanie funkcji
10.  // obliczSumy dla tablicy a o 10 elementach, w miejscu wskaźników z definicji
11.  funkcji znajdują się adresy komórek pamięci, do których będą wstawione
12.   // obliczone wartości
13.  printf("\n sumaP=%d sumaNp=%d sumaD=%d sumaU=%d",\
14.          sumaU, sumaNp, sumaD, sumaU);
15.  system("pause");
16.  return 0;
17.  }
```

```
18.
19. void wczytTab(int x[ ], int n)// definicja funkcji wczytującej tablicę liczb typu int
20. {
21. int i;
22. for (i=0;i<n; i++)
23. {
24. printf("\n Podaj element %d=",i);
25.  scanf("%d",&x[i]);
26. }
27. }
28.
29. int obliczSumy(int x[ ], int n, int *pnp, int *pd, int *pu)// definicja funkcji
30.  // obliczSumy obliczającej dla tablicy x, sumy elementów parzystych,
31.  //nieparzystych , dodatnich i ujemnych i przekazująca obliczoną wartość
32.  // sumy do funkcji wywołującej przy pomocy instrukcji return, pozostałe
33. // wartości z użyciem wskaźników pnp, pd i pu
34.
35. {
36.  int i, sumaP=0, sumaNp=0, sumaD=0, sumaU=0;
37.  for (i=0;i<n; i++)
38. {
39. if (x[i]%2==0) sumaP+=x[i];// parzystość jest sprawdzana poprzez obliczanie
40.  // reszty z dzielenia przez 2. Jeśli reszta jest równa 0, to liczba jest parzysta,
41. // jeśli 1 nieparzysta
42. else  sumaNp+=x[i];
43. if (x[i]>0) sumaD+=x[i];
44. else   sumaU+=x[i];
45. *pnp=sumaNp;// zapis sumy liczb nieparzystych do miejsca pamięci
46. // wskazywanego przez wskaźnik pnp. Rzeczywista wartość wskaźnika jest
47.  ustalana podczas wywołania funkcji poprzez wstawienie na miejsce wskaźnika
48.  konkretnego adresu pamięci
49. *pd=sumaD; // takie samo działanie jak dla wskaźnika pnp
50. *pu=sumaU;
51. }
52. return sumaP;
53. }
```

Gdy tablica wielowymiarowa jest parametrem funkcji, to w standardzie ANSI C konieczne jest podanie wszystkich wymiarów poza pierwszym.

Przykład 75 *Program wczytujący i drukujący dwie tablice o wymiarze 3x3*.

```
1. #include <stdio.h>
2. void wczyt2D( int x[ ][3], int n)// definicja funkcji wczytującej tablicę
3.                                  // dwuwymiarową o n wierszach i 3 kolumnach
4. {
5. int i,j,k;
```

```
6.  for (i=0; i<n;i++)
7.  for (j=0;j<3;j++)
8.  do
9.  {
10.  printf ("\n Podaj element [%d][%d]=",i, j);
11.  k=scanf("%d", &x[i][j]);// wczytywanie elementu tablicy
12.  fflush(stdin);
13.  }
14.  while (k==0);
15.  return;
16.  }
17.
18.  void druk2D( int x[ ][3], int n)//definicja funkcji drukującej  tablicę o n wierszach i 3
19.                                    // kolumnach
20.  {
21.  int i,j,k;
22.  for (i=0; i<n;i++)
23.  {
24.  for (j=0;j<3;j++)
25.  printf ("element [%d][%d]=%d",i, j, x[i][j]);
26.  printf("\n");
27.  }
28.  return;
29.  }
30.
31.  int main()
32.  {
33.  int a[3][3], b[3][3];
34.  wczyt2D(a,3);// wywołanie funkcji wczytującej tablicę a
35.  wczyt2D(b,3);
36.  druk2D(a,3);// wywołanie funkcji drukującej tablicę a
37.  druk2D(b,3);
38.  system("pause");
39.  return 0;
40.  }
```

Przykład 76. *Program kopiujący i drukujący dwie tablice o wymiarze 3x3.*

```
1.  void kopiuj2D(int x[ ][3],int y[][3],int n); //deklaracje funkcji
2.  void druk2D( int x[ ][3], int n);
3.  int main()
4.  {
5.  int b[3][3]= { {2,1,1}, {3,4,2}, {2,3,4}};
6.  int a[3][3];
7.  kopiuj2D( a,b,3);// wywołanie funkcji kopiującej tablicę b do tablicy a
8.  druk2D(a,3); // wywołanie funkcji drukującej tablicę a
9.  system("pause");
```

```
10. }
11.
12. void kopiuj2D( int x[ ][3],int y[ ][3], int n)// definicja funkcji kopiującej tablice
13.                                               // dwuwymiarowe o n wierszach i 3 kolumnach
14. {
15. int i,j,k;
16. for (i=0; i<n;i++)// realizacja kopiowania wierszami tablicy y do tablicy x
17. for (j=0;j<3;j++)
18. x[i][j]=y[i][j];
19.  return;
20. }
21. void druk2D( int x[ ][3], int n)// definicja funkcji drukującej wierszami
22.                                  //tablicę dwuwymiarową o n1 wierszach
23.                                  // i 3 kolumnach
24. {
25. int i,j,k;
26. for (i=0; i<n;i++)
27. {
28. for (j=0;j<3;j++)
29. printf ("element [%d][%d]=%d  ",i, j, x[i][j]);
30. printf("\n");
31. }
32. return;
33. }
```

Przykład 77. *Przepisywanie tablicy dwuwymiarowej do tablicy jednowymiarowej*.

```
1. void kopiuj1D_2D( int b[ ][2],int n1, int d[ ]);// deklaracja funkcji kopiującej
2.                                                  // tablice
3. #include <stdio.h>
4. int main()
5. {
6. int tab2D[2][2]={ {1,2},{2,3}};// definicja i inicjalizacja tablicy dwuwymiarowej
7.                    //o rozmiarze 2 x 2.
8. int tab1D[4];
9. int i;
10. kopiuj1D_2D(tab2D, 2, tab1D);// wywołanie funkcji kopiowania tablic
11. for (i=0;i<4; i++)
12. printf("   %d ", tab1D[i]);
13. system("pause");
14. }
15.
16. void kopiuj1D_2D( int b[ ][2],int n1, int d[ ]) // definicja funkcji kopiującej
17. {
18. int i,j,k=0;
19. for (i=0;i<n1;i++)// przepisywanie elementów tablicy b wierszami do tablicy
20.                    // jednowymiarowej
```

```
21. for (j=0;j<2;j++)
22.  d[k++]=b[i][j];
23. }
```

13 Tablice znaków (łańcuchy)

Tablice znaków stanowią specjalny rodzaj tablic o budowie ułatwiającej przetwarzanie tekstów. Przetwarzanie obejmuje szereg operacji takich jak tworzenie, modyfikacja i przeszukiwanie. Tablice znakowe mogą być definiowane w sposób następujący:

```
char s1[5];
char s2[5 ]="ABCD";
char s3[ ]="abcd";
```

Definicje tablic s2 i s3 zawierają też inicjalizację.

Budowa tablicy znaków

Przykładowo, tablica s2 ma następującą budowę:

'A'	'B'	'C'	'D'	'\0'

Tak więc definiując tablicę, należy przewidzieć jedną komórkę na znak końcowy '\0'.

Stałe łańcuchowe i wskaźniki do znaków

Stałe łańcuchowe (stałe typu tablica znaków), np. "Tekst" , przechowywane są w pamięci jako tablice znaków z ostatnim elementem równym '\0'. "Tekst" jest typu char *, czyli wskaźnik do znaku. Wskaźnik do znaku może więc zostać zainicjowany stałą łańcuchową

```
char  *ps="Tekst";// sposób 1
```

Inicjalizacja taka jest równoważna parze instrukcji

```
char *ps; // sposób 2
ps="Tekst";
```

Warto też zauważyć, że przy definicji tablicy znakowej

```
char s2[5];
```

nie jest możliwa realizacja przypisania w programie

```
s2="ABCD" ;// przypisanie błędne
```

Wynika to z faktu, że s2 jako nazwa tablicy jest typu stały wskaźnik do znaku, który to wskaźnik nie może być zmieniony poprzez przypisanie.

Wczytywanie i drukowanie tablic znakowych

Tablice znakowe mogą być wczytywane przy użyciu funkcji scanf i specjalnego, przeznaczonego do wczytywania tablic deskryptora formatu %s. w sposób następujący:

```
scanf(%s, nazwaTablicyZnakowej);
```

np. dla zdefiniowanej powyżej tablicy znakowej mamy

```
scanf(%s, s1);
```

Przy wczytywaniu tablic znakowych nie ma potrzeby stosowania operatora adresowego &. Jednak wczytywanie łańcuchów przy użyciu funkcji scanf nie pozwala na wczytanie tablic znakowych zawierających spacje, gdyż wczytywanie tablicy znakowej jest przerywane po napotkaniu spacji. Dla tablic takich należy zastosować funkcję gets lub też funkcję fgets. Funkcja gets ma następujący prototyp:

```
char *gets(char *s);
```

Opis działania funkcji gets:

Funkcja wczytuje znaki ze standardowego wejścia (stdin) aż do momentu napotkania znaku nowej linii. Wczytywane znaki są wyświetlane na ekranie i zapamiętywane począwszy od miejsca wskazywanego przez s. Znak nowej linii jest zastępowany w tablicy znakowej znakiem końca tablicy znakowej '\0'. Ciąg wejściowy może zawierać pewne białe znaki (np. spacje i znaki tabulacji poziomej). Funkcja zwraca s, gdy operacja się powiodła, lub NULL w przypadku wystąpienia błędu. Długość wczytywanego ciągu znaków nie jest ograniczana, co przy ciągach dłuższych niż długość argumentu funkcji gets pomniejszonego o 1, może spowodować uszkodzenie sąsiednich obszarów pamięci.

Przykład 78. *Zastosowanie funkcji* gets.

```
1. char lan [20];
2. printf ("\n Podaj tablice znakow:");
3. gets(lan);// wywołanie funkcji wczytującej tablicę znakową
```

Bardziej uniwersalną funkcją służącą do wczytywania tablic znakowych jest fgets. Funkcja fgets ma następujący prototyp:

```
char *fgets(char *s, int n, FILE *stream);
```

Funkcja ta czyta do tablicy znakowej s znaki ze strumienia wejściowego (pliku) określonego przez wskaźnik stream. Funkcja kończy wczytywanie znaków po przeczytaniu *n* -1 znaków lub też po pojawieniu się znaku nowej linii. Funkcja wpisuje znak nowej linii do tablicy znakowej. Funkcja zwraca wskaźnik do tablicy znakowej lub NULL w przypadku pojawienia się końca pliku lub błędu.

Przykład 79. *Wczytywanie tablicy znakowej ze standardowego strumienia wejściowego.*

```
1. char lan [20];
2. printf ("\n Podaj lancuch:");
3. fgets(lan,15,stdin);// funkcja wczytuje ze strumienia stdin( klawiatury) nie więcej
4. // niż 14 znaków lub kończy wczytywanie po podaniu znaku nowej linii
```

Do drukowania tablic znakowych można zastosować funkcję printf lub też funkcje puts i fputs. Przy drukowaniu tablicy znakowej z użyciem printf stosuje się deskryptor %s. Przykładowo, tablicę lan można wydrukować jako

```
printf (" lan=%s", lan);
```

Prototyp funkcji puts ma postać

```
int puts(const char *s);
```

Funkcja wyprowadza tablicę znakową na wyjście standardowe (stdout) i dołącza znak nowej linii. W przypadku pomyślnej realizacji funkcja zwraca wartość nieujemną, w przeciwnym przypadku znak EOF.

Funkcja puts ma prototyp o postaci

```
int fputs(const char *s, FILE *stream);
```

Funkcja zapisuje tablicę znakową s do do strumienia wyjściowego określonego przez wskaźnik stream. Funkcja nie zapisuje do pliku znaku '\0'.W przypadku powodzenia operacji funkcja zwraca wartość nieujemną, w przypadku błędu znak EOF.

Standardowe funkcje łańcuchowe

Poniżej omówione zostaną wybrane, najczęściej spotykane, funkcje związane z przetwarzaniem tablic znakowych(pełny zestaw w string.h i stdlib.h). Funkcje te to: strlen, strcpy, strcat, strcmp, atoi, itoa. Inne ważniejsze funkcje to: strlwr, strupr, strchr, strstr, strtok.

Funkcja strlen

Funkcja ta następujący prototyp:

```
size_t strlen(const char *s);
```

Funkcja oblicza długość tablicy znakowej s bez końcowego znaku '/0'.

Przykład 80. *Użycie* strlen.

```
1. char s[20]="12345678";
2. int dl;
3. dl=strlen(s);// wywołanie funkcji zwracającej długość tablicy znakowej s
```

Funkcja strcpy.

```
char *strcpy(char *dest, const char *src);
```

Kopiuje tablicę znakową src do tablicy znakowel dest. Kopiowanie ulega zakończeniu po skopiowaniu znaku '\0' kończącego tablicę src. Funkcja zwraca wskaźnik dest .

Przykład 81. *Zastosowanie* strcpy.

```
1. #include <stdio.h>
2. #include <string.h>
3. int main(void)
4. {
5. char lan1[10="ABC";
6. char lan2[ ]= "12345";
7. strcpy(lan1, lan2); // wywołanie funkcji strcpy kopiującej tablicę lan1 do tablicy  lan2
8. printf("%s", lan1);
9. return 0;
10. }
```

Funkcja strcat

```
char *strcat(char *dest, const char *src);
```

Dołącza tablicę znakową src do końca tablicy znakowej dest. Funkcja zwraca wskaźnik dest. Długość połączonej tablicy jest równa strlen(dest)+strlen(src).

Przykład 82. *Zastosowanie funkcji* strcat.

```
1. #include <stdio.h>
2. #include <string.h>
3. int main(void)
4. {
5. char lan1[20]="abcde";
```

```
6.  char lan2[ ]= "123456789";
7.  strcat(lan1, lan2); // wywołanie funkcji „sklejającej” tablice znakowe lan1 i lan2,
8.                      // do końca tablicy lan1 dołączana jest tablica lan2
9.  printf("%s", lan1);
10.  return 0;
11.  }
```

Funkcja strcmp

```
int strcmp(const char *s1, const char *s2);
```

Porównuje tablicę znakową s1 z tablicą znakową s2, porównując kody znaków (np. kody ASCII) obu tablic. Porównanie kończy się, gdy w jednej z tablic zostanie napotkany znak o większym kodzie lub też zostanie osiągnięty koniec jednej z tablic (wtedy dłuższa tablica uważana jest za większą). Funkcja zwraca:

-wartość <0, gdy s1<s2,
-wartość 0, gdy s1==s2,
-wartość >0, gdy s1>s2.

Przykład 83. *Zastosowanie funkcji* strcmp.

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <conio.h>
4.  int main(void)
5.  {
6.  char lan1[10]="abcde", lan2[10 ]; int p;
7.  printf(" Podaj lan2:");
8.  scanf("%s", lan2);
9.  p= strcmp(lan1,lan2);// wywołanie funkcji porównującej tablice znakowe lan1 i lan2
10.  if (p<0) printf(" lan1<lan2");
11.  else if ( p==0) printf("\nlan1=lan2");
12.  else printf("\n lan1>lan2");
13.  getch();
14.  return 0;
15.  }
```

Funkcja atoi

```
int atoi(const char *s);
```

Przekształca tablicę znakową s na liczbę całkowitą.

Tablica znakowa s może zawierać
- opcjonalny łańcuch tabulacji i spacji

- opcjonalny znak
- łańcuch cyfr

Pierwszy nierozpoznany znak kończy konwersję.

Wartość zwracana: atoi zwraca liczbę odpowiadającą przekształconej tablicy znakowej. Jeśli tablica znakowa nie może być przekształcona, funkcja zwraca 0.

Przykład 84. Konwersja tablicy znakowej na liczbę przy zastosowaniu atoi.

```
1. #include <stdlib.h>
2.  #include <stdio.h>
3. int main(void)
4. {
5. int liczba;
6. char *str = "12345";// definicja wskaźnika do znaku i inicjalizacja wskaźnika
7.                      //stałą łańcuchową

8. liczba = atoi(str);   // wywołanie funkcji zamieniającej tablicę znakową na liczbę
                         //całkowitą
9. printf("Tablica znakowa= %s liczba = %d", str, n);
10. return 0;
11. }
```

Funkcja itoa

```
char *itoa(int liczba, char *tablicaZnakowa , int podstawa);
```

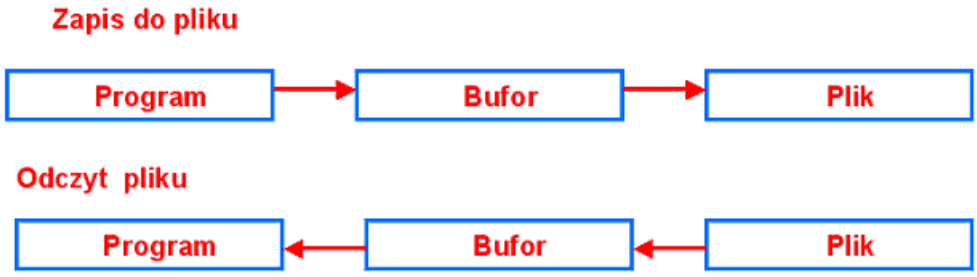
Zamienia liczbę całkowitą na tablicę znakową. Parametr podstawa określa podstawę systemu liczbowego stosowanego przy zamianie, podstawa musi się zawierać w zakresie od 2 do 36. Jeśli wartość jest ujemna i podstawa jest równa 10, pierwszym znakiem tablicy znakowej jest '-'.

Przykład 85. Konwersja liczby całkowitej na tablicę znakową przy zastosowaniu funkcji itoa.

```
1. #include <stdlib.h>
2. #include <stdio.h>
3. int main(void)
4. {
5. int liczba = 12345;
6. char tablicaZnakowa[25];
7. itoa(liczba, tablicaZnakowa, 10);  // wywołanie funkcji itoa zamieniającej liczbę 12345 na
8. //tablicę znakową dla podstawy systemu liczbowego  równej 10
9. printf("liczba = %d  Tablica znakowa = %s\n", liczba, tablicaZnakowa);
10. return 0;
11. }
```

14 Pliki

Plik jest podstawową jednostką służącą do przechowywania informacji w każdym systemie operacyjnym. Każdy plik ma swoją nazwę, długość w bajtach, atrybuty określające sposób korzystania z pliku oraz nadane uprawnienia dla poszczególnych użytkowników określające sposób korzystania z pliku. W systemach komputerowych występują różnego rodzaju urządzenia, do których przesyłane są dane czy też są z nich pobierane(np. drukarki, pliki dyskowe, porty komunikacyjne). Z punktu widzenia programu takie urządzenia traktowane są też jako pliki. System operacyjny izoluje programy użytkowe od szczegółów pracy poszczególnych urządzeń. Jest to realizowane przy użyciu tzw. strumieni. Strumienie stanowią element pośredniczący między programem a systemem operacyjnym. Wyróżniamy strumienie binarne, które stanowią ciągi bajtów bez żadnej specjalnej struktury. Plik związany ze takim strumieniem jest to tzw. plik binarny. Strumień może też posiadać pewną strukturę – składać się z linii tekstu zakończonych znakiem końca linii. Strumień taki nazywamy strumieniem tekstowym, a plik związany z takim strumieniem to tzw. plik tekstowy. Strumienie są zwykle buforowane, oznacza to, że np. zapis w programie do pliku jest tylko zapisem do pewnego obszaru pamięci. Zawartość bufora jest zapisywana na nośniku, gdy bufor zostanie napełniony lub też program zażąda tego w sposób jawny. Długość pliku może być ograniczona wielkością nośnika lub też ograniczenie może wynikać z pewnych własności systemowych (zakres typu całkowitego).



Elementy pliku są zwykle dostępne sekwencyjnie, co oznacza, że w danym momencie istnieje dostęp tylko do jednego elementu pliku. Liczba elementów pliku może się zmieniać w trakcie przetwarzania pliku. Koniec pliku jest zaznaczany z punktu widzenia języka C znakiem EOF(ang. end-of-file). W pliku jest to znak ctrl-Z (ASCII 26). Z każdym plikiem jest związana pewna zmienna, tzw. wskaźnik pliku określająca aktualnie dostępny element pliku. Numeracja elementów pliku zaczyna się od zera.

Identyfikacja pliku w systemie operacyjnym może się odbywać poprzez wskaźnik do pliku (ang. file pointer) lub też uchwyt pliku (ang. handle) będący pewną liczbą identyfikującą plik w ramach systemu operacyjnego.

Język C nie ma wbudowanych operacji wejścia-wyjścia związanych z realizacją funkcji plikowych, stosowane są dla ich realizacji funkcje biblioteczne z pliku <stdio.h>. Plik ten zawiera prototypy funkcji wejścia/wyjścia oraz definicje trzech typów: size_t, fpos_t oraz FILE. Typy size_t, fpos_t to odmiany typu unsigned int. Struktura FILE zawiera różne dane związane z plikiem, w tym nazwę pliku, jego status i aktualną pozycję w pliku.

Aby rozpocząć realizację operacji plikowych na istniejącym pliku lub też utworzyć nowy plik, należy zdefiniować wskaźnik do struktury FILE, fp w sposób następujący:

```
FILE *fp;
```

Należy odróżniać wskaźnik fp, od wskaźnika pliku – zmiennej wewnętrznej wskazującej, gdzie w pliku w danym momencie można zapisywać dane lub odczytywać.

Przetwarzanie pliku w trakcie działania programu może polegać na wykonywaniu różnego rodzaju operacji na pliku np. utworzeniu lub otwarciu pliku, zapisie elementów, odczycie elementów, usuwaniu elementów czy też porządkowaniu (np. sortowaniu). Jednak zanim rozpocznie się działania na pliku należy plik utworzyć(jeśli nie istniał) albo też otworzyć, jeśli istniał.

Pliki - tworzenie i otwieranie

W języku C plik jest tworzony lub otwierany przy użyciu funkcji fopen (istnieje też funkcja open do tzw. operacji niskopoziomowych). Po zakończeniu operacji na pliku, plik jest zamykany przy użyciu funkcji fclose. Zamknięcie pliku powoduje zarejestrowanie zmian dokonanych w pliku w trakcie jego przetwarzania.

Funkcja fopen i parametry otwarcia pliku

```
FILE * fopen( const char*nazwa_pliku, const char *tryb);
```

Funkcja fopen posiada dwa parametry. Pierwszy parametr nazwa_pliku jest tablicą znakową określającą nazwę pliku, drugi parametr tryb też będący tablicą znaków, określa tzw. tryb utworzenia lub otwarcia pliku.

Przykład 86.*Tworzenie i otwieranie nowego pliku.*

```
FILE *fp;
fp=fopen("Plik1.dat", "w");
```

Nazwa pliku podana bez ścieżki dostępu oznacza, że plik będzie się znajdował w tym katalogu, w którym zapisano projekt. Można też podać ścieżkę dostępu do pliku w sposób następujący:

```
fp=fopen("C:\\moje dokumenty\\Plik2.dat", "w");
```

(należy stosować podwójny ukośnik dla uzyskania pojedynczego znaku \, dla znaku tego wymagana jest sekwencja ucieczki).

Funkcja fopen zwraca wskaźnik do pliku, jeśli jego otwarcie lub utworzenie było możliwe. Jeżeli nie było to możliwe, funkcja fopen zwraca zerowy wskaźnik NULL. Wskazanym jest sprawdzanie

uzyskanej wartości fp przed rozpoczęciem operacji plikowych w następujący sposób:

Przykład 87. *Tworzenie nowego pliku*.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.  int main ()
4. {
5. FILE *fp;
6. fp=fopen("Plik1.dat", "w");
7. if (fp!=NULL) { //wystarczy if (fp)
8. printf("Plik został utworzony\n");
9. system("pause");
10. }
11.  else
12. {
13. printf(" Plik nie zostal utworzony\n");
14. system("pause");
15. exit(1);
16. }
17. }
```

Tabela. Tryby tworzenia i otwierania pliku.

Tryb	Jeśli plik istnieje	Jeśli plik nie istnieje
"r"	Otwiera plik do odczytu	Błąd
"w"	Otwiera nowy plik, jeśli istniał plik o danej nazwie jego zawartość jest kasowana	Tworzy nowy plik
"a"	Otwiera plik do dopisywania	Tworzy nowy plik
"r+"	Otwiera plik do odczytu i zapisu	Błąd
"w+"	Otwiera nowy plik do zapisu i odczytu, jeśli istniał plik o danej nazwie jego zawartość jest kasowana	Tworzy nowy plik
"a+"	Otwiera plik do dopisywania i odczytu	Tworzy nowy plik

Powyższe tryby dotyczą zasadniczo plików tekstowych (zależy to od ustawienia zmiennej systemowej _fmode, może ona przybierać wartości O_BINARY lub O_TEXT, standardowo ma wartość O_TEXT). Dla plików binarnych należy dodać do nazwy trybu literę b, np. "rb", "wb", "r+b", "w+b". Dla plików tekstowych należy dodać literę t, czyli "rt", "wt", "r+t", "w+t", jeśli zmienna_fmode ma wartość O_BINARY.

Podstawowe funkcje plikowe ANSI C

clearerror	fgetpos	fread	getc	puts	tmpfile
fclose	fgets	fscanf	getchar	remove	tempnam
feof	fopen	fseek	gets	rename	ungetc
ferror	fprintf	fsetpos	perror	rewind	
fflush	fputc	ftell	putc	setbuf	
fgetc	fputs	fwrite	putchar	setvbuf	

Funkcja fopen (uzupełnienie)

```
FILE *fopen(char const *nazwa_pliku, char const *tryb);
```

Argument pierwszy nazwa_pliku wskazuje plik do otwarcia, argument drugi tryb określa, czy plik (strumień) zostanie otwarty do zapisu, odczytu czy obu tych operacji. Określa on też tryb (binarny czy tekstowy). Funkcja zwraca wskaźnik do pliku (wskaźnik do struktury FILE wykorzystywanej przy dostępie do strumienia) albo też NULL. Maksymalną liczbę otwartych plików określa stała FOPEN_MAX, maksymalną długość nazwy stałą FILENAME_MAX.

Funkcja fclose

```
int fclose (FILE *stream);
```

Funkcja zamyka strumień wskazywany przez parametr FILE. Funkcja zapisuje buforowane dane strumienia stream, zwalnia wszelkie automatycznie alokowane bufory i zamyka strumień. Funkcja zwraca 0, jeśli działanie się powiodło lub też EOF. Przykładowe jej użycie to: fclose(fp);
gdzie fp jest wskaźnikiem do otwartego pliku.

Funkcja ftell

```
long int ftell(FILE *stream);
```

Zwraca pozycję wskaźnika pliku wskazywanego przez *stream. Dla plików binarnych pozycja jest mierzona w bajtach od początku pliku. Jeżeli powstał błąd, zwraca -1L i nadaje zmiennej globalnej errno wartość dodatnią. errno jest ustawiana na jedną z dwóch wartości :

BADF - niewłaściwy wskaźnik do pliku

ESPIP - niedopuszczalna operacja przesuwania wskaźnika pliku na urządzeniu

Funkcja fseek

```
int fseek(FILE *stream, long int offset,int whence);
```

Funkcja ustawia nową pozycję bieżącą pliku binarnego wskazywanego przez stream. Następna operacja plikowa rozpocznie się od tej pozycji. Nowa pozycja jest oddalona o offset bajtów od punktu odniesienia określonego przez trzeci parametr whence. Parametr ten może przyjmować następujące wartości reprezentowane przez makra:

SEEK_SET – przesunięcie jest realizowane względem początku pliku
SEEK_CUR – przesunięcie jest realizowane względem bieżącej pozycji
SEEK_END – przesunięcie jest realizowane względem końca pliku

Powyższym stałym odpowiadają wartości liczbowe: 0, 1, 2. Funkcja zwraca wartość różną od zera w przypadku wystąpienia błędu. Parametr offset dla parametru whence=SEEK_SET musi być nieujemny. Próba przesunięcia przed początek pliku jest błędem. Dla whence=SEEK_CUR offset może mieć dowolny znak, dla whence=SEEK_END przesunięcie poza koniec pliku i zapis danych powodują powiększenie pliku, przesunięcie poza koniec pliku i odczyt powoduje zwrócenie wartości EOF. Dla strumieni tekstowych parametr offset musi mieć wartość 0, jeśli whence ma wartość SEEK_CUR lub SEEK_END. Dla whence=SEEK_SET parametr offset musi mieć wartość równą wartości zwróconej przez funkcję `ftell` dla tego strumienia. Przykładowo

fseek(fp,0, SEEK_SET) – oznacza przesunięcie wskaźnika pliku na początek pliku,

fseek(fp,4, SEEK_SET) – oznacza przesunięcie wskaźnika pliku o 4 bajty względem początku pliku

fseek(fp,8, SEEK_CUR) – oznacza przesunięcie wskaźnika pliku o 8 bajtów względem jego aktualnej pozycji w kierunku końca pliku

fseek(fp,0, SEEK_END) – oznacza przesunięcie wskaźnika pliku na koniec pliku,

fseek(fp,-12, SEEK_END) – oznacza cofnięcie wskaźnika pliku o12 bajtów względem końca pliku

Funkcja fwrite

```
size_t fwrite(void *buffer, size_t size, size_t count, FILE *stream );
```

Funkcja zapisuje do pliku reprezentowanego przez stream z miejsca pamięci traktowanego jako tablica wskazywanego przez buffer, count bloków danych, każdy o rozmiarze size. Funkcja zwraca liczbę zapisanych bloków. Jest ona mniejsza niż count w przypadku wystąpienia błędu.

Przykładowe wywołanie funkcji może mieć postać

```
fwrite(&x, sizeof( double), 1, fp);
```

Oznacza ono instrukcję zapisu wartości zmiennej x do pliku wskazywanego przez wskaźnik fp. Poszczególne argumenty oznaczają: &x – adres zmiennej x, sizeof(double) rozmiar w bajtach zmiennej typu double, argument trzeci – liczba 1, oznacza liczbę bloków danych zapisywanych do pliku w danym wywołaniu funkcji fwrite, fp jest wskaźnikiem do pliku, do którego odbywa się zapis. Operator sizeof(int) jest operatorem zwracającym rozmiar swojego argumentu w bajtach.

Funkcja fread

```
size_t fread(void *buffer,size_t size,size_t count, FILE *stream );
```

Funkcja odczytuje z pliku reprezentowanego przez stream do miejsca pamięci wskazywanego przez buffer, count bloków danych, każdy o rozmiarze size. Funkcja zwraca liczbę odczytanych bloków. Jest ona mniejsza niż count w przypadku wystąpienia błędu. Wystąpienie błędu lub dojście do końca pliku można wykryć funkcjami ferror i feof.

Przykładowe wywołanie funkcji może mieć postać

```
fread(&liczba, sizeof(int), 1, fp);
```

Oznacza ono instrukcję odczytu jednej wartości typu int z pliku wskazywanego przez wskaźnik fp i zapis tej wartości do zmiennej liczba. Poszczególne argumenty oznaczają: &liczba – adres zmiennej liczba, sizeof(int) rozmiar w bajtach zmiennej typu int, argument trzeci – liczba 1, oznacza liczbę bloków odczytywanych z pliku w danym wywołaniu funkcji fread, fp jest wskaźnikiem do pliku, z którego odbywa się odczyt.

Przykład 88. *Program zapisujący 3 liczby typu int do pliku i następnie je odczytujący.*

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  int main()
4.  {
5.  int i1=5,i2=10,i3=15,buf;
6.  FILE *fp;
7.  fp=fopen("Plik1.dat","w+b");// utworzenie  w katalogu bieżącym pliku o nazwie Plik1.dat
8.  // wskazywanego przez fp w trybie  ”w+b” oznaczającym, że zostanie utworzony
9.  // nowy plik z możliwością zapisu i odczytu danych
10.  if (fp==NULL) // sprawdzenie czy operacja się powiodła, jeśli tak, wskaźnik fp
11.  // zwrócony przez funkcję fp ma wartość różną od zera, jeśli utworzenie nie
12.  // nie było możliwe zwracana jest wartość zero, reprezentowana tutaj przez
13.  // stałą wskaźnikową NULL.
14.  {
15.   printf("\n Plik nie daje sie utworzyc");
16.   system("pause");
17.   return 1; // zakończenie działania programu, jeśli pliku nie dało się utworzyć
```

```
18.  }
19.  fwrite(&i1,sizeof(int),1,fp);// zapis wartości zmiennej i1 do pliku
20.  fwrite(&i2,sizeof(int),1,fp);// zapis i2
21.  fwrite(&i3,sizeof(int),1,fp);// zapis i3
22.  fseek(fp,0,SEEK_SET);       // ustawienie wskaźnika pliku na początek pliku
23.  fread(&buf,sizeof(int),1,fp); // odczyt pierwszej wartości z pliku i zapis tej
24. //  wartości do zmiennej buf
25.  printf("\n Element nr 1 =%d",buf);
26.  fread(&buf,sizeof(int),1,fp); // odczyt drugiej wartości z pliku
27.  printf("\n Element nr 2 =%d",buf);
28.  fread(&buf,sizeof(int),1,fp); // odczyt trzeciej wartości z pliku
29.  printf("\n Element nr 3 =%d\n",buf);
30.  fclose(fp); // zamknięcie pliku
31.  system("pause");
32.  return 1;
33.  }
```

Przykład 89. *Program zapisujący tablicę liczb typu double do pliku i następnie ją odczytujący.*

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  int main()
4.  {
5.  int i;
6.  double  buf, tab[5]={ 1.5,-1,2,3.2,5.1};// inicjalizacja tablicy
7.  FILE *fp;
8.  fp=fopen("Plik2.dat","w+b");
9.  for (i=0;i<5;i++) // Zapis 5 liczb do pliku
10.  fwrite(&tab[i],sizeof(double),1,fp);
11.  fclose(fp); // zamiast zamykania i ponownego otwierania  można użyć  fseek(fp,0,0)
12.  //lub rewind.
13.  fp=fopen("Plik2.dat","r+b");
14.  for (i=0;i<5;i++)// Odczyt 5 liczb z pliku
15.  {
16.  fread(&buf,sizeof(double),1,fp);
17.  printf("\n Element pliku nr %d =%5.2f",i,buf);
18.  }
19.  fclose(fp);
20.  getch();
21.  return 0;
22.  }
```

Przykład 90. *Program zapisujący tablicę liczb typu* double *do pliku i następnie ją odczytujący -wersja II, użycie zapisu i odczytu blokowego dla kilku bloków.*

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main()
4. {
5. int i;
6. double  tabBuf[5], tab[5]={ 1.5,-1,2,3.2,5.1};
7. FILE *fp;
8. fp=fopen("Plik3.dat","w+b");
9. fwrite(tab,sizeof(double),5,fp);// zapis 5 liczb do pliku z zastosowaniem
10. // operacji zapisu blokowego, pierwszy argument będący nazwą tablicy,
11. // oznacza adres początku obszaru pamięci, z którego pobierane są dane do
12. // zapisu, drugi argument rozmiar  pojedynczego bloku w bajtach, trzeci argument –
13. // liczba 5, oznacza liczbę  zapisywanych bloków, każdy o rozmiarze sizeof(double), a
14. // czwarty argument wskaźnik do pliku, do którego odbywa się zapis.
15. //
16.  fclose(fp);
17. }
18. fp=fopen("Plik2.dat","r+b");
19. // Odczyt 5 liczb z pliku
20. fread(tabBuf,sizeof(double),5,fp); //odczyt 5 liczb z pliku z zastosowaniem
21. // operacji odczytu blokowego, pierwszy argument będący nazwą tablicy,
22. // oznacza adres początku obszaru pamięci, do którego zapisywane są dane odczytane
23. // z pliku, drugi argument rozmiar  pojedynczego odcztywanego bloku w bajtach, trzeci
24.  //argument –  liczba 5, oznacza liczbę  odczytywanych  bloków, każdy o rozmiarze
25. // sizeof(double), a czwarty argument wskaźnik do pliku, z którego odbywa się odczyt.
26. for (i=0;i<5;i++)
27. printf("\n Element pliku nr %d =%5.2f",i,tabBuf[i]);
28. fclose(fp);
29. getch();
30. return 0;
31. }
```

Wyznaczanie rozmiaru pliku

Rozmiar pliku jest mierzony w bajtach. Bajt jest najmniejszą jednostką zapisywaną do pliku lub odczytywaną z pliku. Przy wyznaczaniu długości można wykorzystać funkcję ftell zwracającą bieżącą pozycję wskaźnika pliku. Jeśli wskaźnik pliku jest ustawiony na końcu pliku, funkcja ftell podaje liczbę bajtów w pliku, czyli rozmiar pliku.

Algorytm. Wyznaczanie długości pliku (plik otwarty).

1. Ustaw wskaźnik pliku na końcu pliku.
2. Wywołaj funkcję ftell.

Przykład 91. *Wyznaczanie długości pliku*.

```
1. #include <stdio.h>
2. #include <conio.h>
3.
4. int main()
5. {
6. int i, liczbaBajtow, liczbaElementow;
7. double  x[5]={ 2.3,0,22,1.2,5.3};
8. FILE *fp;
9. fp=fopen("Plik3.dat","r+b");// otwarcie istniejącego pliku w trybie do odczytu
10. // i zapisu, plik musi znajdować się w katalogu bieżącym ( tam gdzie zapisano
11. // projekt), powinno się tutaj zamieścić sprawdzanie powodzenia operacji,
12. // jak w przykładzie 89. Ze względu na konieczność ograniczenia rozmiaru
13. // poszczególnych przykładów, sprawdzanie to jest pomijane.
14. fseek(fp,0,SEEK_END);// ustawienie wskaźnika pliku na końcu pliku
15. liczbaBajtow=ftell(fp);// odczyt aktualnej pozycji wskaźnika pliku
16. liczbaElementow=liczbaBajtow/sizeof(double); // wyznaczenie liczby elementów
17. // typu double znajdujących się w pliku
18. printf("\n Liczba bajtow:%d",liczbaBajtow);
19. printf("\n Liczba elementow :%d",liczbaElementow);
20. fclose(fp);
21. }
```

Przykład 92. *Obliczanie w pliku sum elementów dodatnich, ujemnych i ilości elementów zerowych.*

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main()
4. {
5. double sumD=0,sumU=0,ilZer=0,buf;
6. FILE *fp;
7. fp=fopen("Plik3.dat","r+b");
8. // wersja 1
9. fseek(fp,0,SEEK_END);// ustawienie wskaźnika pliku na końcu pliku
10. iloscElementow=ftell(fp)/sizeof(double);// wyznaczenie liczby elementów typu
11. // double znajdujących się w pliku
12. printf("\n Ilosc elementow pliku=%d",iloscElementow);
13. fseek(fp,0,SEEK_SET); /// ustawienie wskaźnika pliku na początku pliku
14. for (i=0;i< iloscElementow;i++)// pętla realizująca powtarzanie operacji odczytu
15. {
16. fread(&buf,sizeof(double),1,fp);// odczyt pojedynczej wartości typu double z
17. // pliku i zapis do zmiennej buf.
18. if (buf>0) sumaD=sumaD+buf;
19. if (buf<0) sumaU=sumaU+buf;
20. if (buf==0) ilZer++;
21. }
```

```
22.  printf("\n Suma el. dodatnich :%5.2f",sumaD);
23.  printf("\n Suma el. ujemnych  :%5.2f",sumaU);
24.  printf("\n Ilosc  zer  :%5.2f",ilZer);
25.
26.  // wersja 2 bez wyznaczania dlugości pliku
27.  fseek(fp,0,SEEK_SET);
28.  while( fread(&buf,sizeof(double),1,fp))//pętla ta realizuje odczyt pliku bez
29. // uprzedniego wyznaczania liczby elementów. Jeśli funkcja fread odczyta blok
30. // z pliku, zwraca wartość niezerową w miejsce swego wywołania w pętli
31. //  while, wartość dla pętli reprezentuje wartość logiczną prawda, co powoduje
32. // przejście do kolejnego wykonania treści pętli. Jeśli natomiast wszystkie
33. // elementy pliku zostaną odczytane, wskaźnik pliku znajdzie się na końcu
34. // pliku, funkcja fread nie będzie mogła odczytać kolejnego i zwróci 0. Wartość
35.  // ta zostanie potraktowana w wyrażeniu sterującym pętlą while, jako wartość
36. // logiczna fałsz i pętla ulegnie zakończeniu
37.  {
38.  if (buf>0) sumaD+=buf;
39.  if (buf<0) sumaU+=buf;
40.  if (buf==0) ilZer++;
41.  }
42.  printf("\n Suma el. dodatnich :%5.2f",sumaD);
43.  printf("\n Suma el. ujemnych  :%5.2f",sumaU);
44.  printf("\n Ilosc  zer  :%5.2f",ilZer);
45.  fclose(fp);
46.  getch();
47.  return 0;
48.  }
```

Przykład 93. *Program wyszukujący w pliku wartość minimalną i maksymalną (zastosowany algorytm jest taki jak w przykładzie 51 dla tablic)*

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  int main()
4.  {
5.  int i;
6.  double max,min,buf;
7.  FILE *fp;
8.  fp=fopen("Plik2.dat","rb");// otwarcie istniejącego pliku w trybie do odczytu
    fread(&buf,sizeof(double),1,fp);// odczyt pierwszego elementu pliku

9.  max=min=buf;        // pierwszy element pliku jest traktowany jako początkowa
10.                        // wartość zmiennych przechowujących w trakcie realizacji
11.                       // algorytmu wartości minimalną i maksymalną
12.  while( fread(&buf,sizeof(double),1,fp))// pętla realizująca odczyt kolejnych
13.   //elementów pliku, począwszy od drugiego , działanie tej konstrukcji jest
14. // omówione w przykładzie 93/
15.  {
```

```
16.  if (buf>max) max=buf; // jeśli wartość odczytanego elementu z pliku znajdująca
17.                        // jest większa od wartości maksymalnej wyznaczonej do
18.                        //danego momentu, wartość ta jest przyjmowana
19.                        //za wartość maksymalną
20.
21.  if (buf<min) min=buf;// jeśli wartość odczytanego elementu z pliku znajdująca
22.                        // jest mniejsza od wartości minimalnejwyznaczonej do
23.                        //danego momentu, wartość ta jest przyjmowana
24.                        //minimalną
25.  }
26.  printf("\n Wartosc maksymalna=%5.2f",max);
27.  printf("\n Wartosc minimalna= %5.2f",min);
28.  getch();
29.  fclose(fp);
30.  return 0;
31.  }
```

Przykład 94. *Program dopisujący do pliku zadaną liczbę elementów typu* int.

```
1.
2.  #include <stdio.h>
3.   #include <conio.h>
4.   int main()
5.  {
6.  int i, buf1,buf2, buf,n=3;
7.  FILE *fp;
8.  fp=fopen("Plik2.dat","a+b");// otwarcie pliku w trybie do dopisywania na końcu pliku i do odczytu.
9.  for (i=0;i<n;i++) // wczytywanie n liczb z klawiatury i dopisywanie na końcu pliku
10.  {
11.  printf("\n Podaj liczbe:");
12.  scanf("%d",&buf);
13.  fwrite(&buf,sizeof(int),1,fp);
14.  }
15.  fseek(fp,0,0);// ustawienie wskaźnika na początku pliku
16.  while(fread(&buf,sizeof(int),1,fp))// pętla realizująca odczyt elementów z pliku
17.  printf("\n Element pliku nr %d =%d",i,buf);
18.  getch();
19.  return 0;
20.  }
21.
```

Przykład 95. *Program kopiowania zawartości pliku.*

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.   int main()
4.  {
5.  int i=0, buf,n=3;
```

```
6.  FILE *fp1,*fp2;
7.  fp1=fopen("Plik2.dat","rb");// otwarcie, w trybie do odczytu, pliku, którego
8.                                 // zawartość będzie kopiowana
9.
10.  fp2=fopen("Plik3.dat","w+b"); // utworzenie nowego pliku, do którego będzie
11.                                    // kopiowana zawartość pliku Plik2.dat
12.  while(fread(&buf,sizeof(char),1,fp1))// pętla odczytująca zawartość pierwszego
13. //pliku bajt po bajcie (dlatego zastosowano jednobajtowy typ char)
14.  fwrite(&buf,sizeof(char),1,fp2);// zapis odczytanego bajtu do drugiego pliku.
15.  fseek(fp2,0,SEEK_SET);
16.  while(fread(&buf,sizeof(int),1,fp2))// Odczyt i wydruk zawartości skopiowanego
17.                                          // pliku
18.  printf("\n Element pliku nr %d =%d",i++,buf);
19.  fclose(fp1); fclose(fp2);
20.  getch();
21.  return 0;
22.  }
23.  }
```

Przykład 96. *Zapis do pliku dwóch tablic znakowych i odczyt.*

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.   int main(int argc, char **argv)
6.  {
7.  char s1[10]="ABCDE";
8.  char s2[10]="abcde",
9.  char  s3[10], s4[10];
10.  FILE *fp;
11.  fp=fopen("Plik1.txt","w+");// utworzenie i otwarcie pliku tekstowego w trybie do
12.                                  //zapisu i odczytu
13.  fputs(s1,fp);// zapis do  pliku tablicy znakowej s1
14.  fputc('\n',fp);// zapis do pliku znaku nowej linii
15.  fputs(s2,fp);// zapis do pliku tablicy znakowej s2
16.  fputc('\n',fp);  // dopisanie znaku nowej linii, aby funkcja
17.                  // fgets czytała tylko pierwszy łańcuch z pliku
18.  fseek(fp,0,0);      // ustawienie wskaźnika pliku na początku pliku
19.  fgets(s3,10,fp);    // wczytanie z pliku pierwszej linii tekstu i zapis do tablicy
20.                      // znakowej s3, funkcja przerywa wczytywanie po napotkaniu
21.                      // znaku nowej linii '\n'
22.  printf("%s",s3);    // wyprowadzenie na ekran pierwszej linii
23.  fgets(s4,10,fp);    // wczytanie drugiej linii z pliku
24.  printf("%s",s4);    // wyprowadzenie na ekran drugiej linii
25.  getch();
26.  fclose(fp);
```

```
27. return 0;
28. }
```

Funkcja fprintf

```
int fprintf(FILE *stream, const char *format,...);
```

Funkcja fprintf wypisuje do strumienia wskazywanego przez parametr stream wartości argumentów wymienionych na liście argumentów zgodnie z parametrem format. Zwracana wartość jest równa ilości wypisanych znaków. Jeśli wystąpi błąd, zwracana jest wartość ujemna.

Przykład 97. *Zastosowanie funkcji fprintf do zapisania dwóch liczb do pliku.*

```
FILE *fp;
fp=fopen("Plik.txt","w")
fprintf(fp,"Test fprintf  %d %f", 2, 24.2);
```

Funkcja fscanf

```
int fscanf(FILE *stream, const char *format,...);
```

Funkcja działa podobnie jak scanf, lecz dane pobiera nie ze standardowego strumienia wejściowego stdin (zwykle klawiatury), ale ze strumienia wskazywanego przez parametr stream. Dane są pobierane zgodnie z parametrem format. Funkcja zwraca liczbę argumentów, którym udało się przypisać wartości.

Przykład 98. *Zastosowanie funkcji* fscanf *do odczytu trzech liczb z pliku.*

```
FILE *fp;
int x;
float y;
double z;
p=fopen("Plik.txt","r")
 fscanf(fp,"%d%f%lf",&x,&y,&z);
```

15 Struktury

Struktury (zwane w innych językach programowania rekordami) stanowią typ służący do przetwarzania informacji powiązanych ze sobą. Definicja typu strukturowego ma następującą postać

```
struct nazwaTypu
```

```
{
        typPola1   nazwaPola1;
        typPola2   nazwaPola2;
        .
        .
        typPolaN  nazwaPolaN;
};
```

Tak więc definicja typu strukturowego składa się ze słowa kluczowego struct, po którym następuje nazwa danego typu strukturowego(czasami określana jako etykieta), po czym po nawiasie otwierającym występują typy poszczególnych składowych struktury oraz ich nazwy. Składowe zwane są polami struktury. Definicja typu strukturowego kończy się nawiasem zamykającym i średnikiem. W typie strukturowym mogą wystąpić pola, które same mogą być typu strukturowego, mówimy wtedy o tzw. typie strukturowym zagnieżdżonym. Nazwy pól muszą być różne w ramach jednego typu strukturowego, jednak można stosować te same nazwy w ramach innych typów strukturowych. Nazwa pola struktury może być taka sama jak nazwa danego typu strukturowego czy też zmiennej strukturowej.

Przykład 99. *W ramach jednego typu danych chcemy przechowywać pewne informacje o pracownikach. Zdefiniujemy uproszczony typ strukturowy pozwalający na przechowywanie takich informacji (w rzeczywistych sytuacjach może być konieczne stosowanie nawet kilkudziesięciu pól).*

```
1. struct pracownik
2. {
3. char nazwisko [20];
4. char imie [20];
5. int wiek;
6. double dochod;
7. };
```

. Zmienne strukturowe można zdefiniować w sposób następujący:

```
struct pracownik pracownik1, pracownik2;
```

Zmienne strukturowe mogą być też zdefiniowane bezpośrednio po nawiasie zamykającym definicję typu strukturowego. Tablica struktur definiowana jest w sposób następujący:

```
struct pracownik pracownicy [20];
```

Zmienna pracownicy jest tablicą struktur składającą się z 20 struktur typu pracownik.

Dla realizacji dostępu do pól typu strukturowego wprowadzono w języku C specjalny operator dostępu do pól struktury – operator " ." (kropka). Operator ten jest stosowany w sposób

następujący:

```
nazwaZmiennejStrukturowej.nazwaPolaStruktury
```

W ogólnym przypadku, polami zmiennych strukturowych możemy posługiwać się jak zmiennymi takiego typu, jak typ danego pola struktury.

Wprowadzanie/wyprowadzanie informacji z /do pól zmiennych strukturowych

Sposób wprowadzania wartości do pól zmiennych strukturowych zależy od typu pola. Jak zaznaczono powyżej, pól struktury można używać jak zmiennych danego typu. dla powyżej zdefiniowanego typu pracownik i zdefiniowanych zmiennych.

Do pól typów numerycznych wartości można wprowadzać stosując instrukcję przypisania albo też instrukcję wczytywania ze standardowego strumienia wejściowego stdin (klawiatury) lub z pliku.

1. Pola numeryczne

```
pracownik1.dochod=1000;// przypisanie wartości 1000 polu dochod
                       //zmiennej strukturowej pracownik1
prcownik1.wiek=25;     // przypisanie wartości 25 polu wiek  zmiennej
                       //strukturowej pracownik1
pracownicy[0].dochod=1200; // przypisanie wartości 1200 polu dochod pierwszego
                           //elementu tablicy struktur pracownicy
pracownicy [0].wiek=24;  // przypisanie wartości 24  polu wiek pierwszego
                         //elementu tablicy struktur pracownicy

scanf("%lf",&pracownik2.dochod);// wczytanie wartości z klawiatury i zapis do
                                //  pola dochod  zmiennej strukturowej pracownik1

scanf("%d",&pracownik2.wiek);   // wczytanie wartości z klawiatury i zapis do
                                //  pola wiek  zmiennej strukturowej pracowni
```

2. Pola typu tablica znaków

```
strcpy (pracownik1.nazwisko,"Kowalski");// kopiowanie nazwiska Kowalski do pola
                                        // nazwisko zmiennej strukturowej pracownik1

strcpy (pracownicy[0].nazwisko,"Janowski");// kopiowanie nazwiska Janowski do
                                           // pola  nazwisko zmiennej strukturowej
                                           //pracownik1
scanf("%s", pracownik2.nazwisko); //wczytanie  z klawiatury tablicy znakowej i zapis
do
                                //  pola nazwisko zmiennej strukturowej pracownik2
gets( pracownik3.nazwisko);
```

Wyprowadzanie informacji odbywa się podobnie jak dla innych zmiennych.

```
printf ("\n   %d ", pracownik1.dochod);
```

```
printf("\n    %s ", pracownik2.nazwisko);
```

Przykład 100. *Inicjalizacja tablicy struktur typu* student.

```
1.  struct student // definicja typu strukturowego student
2.  {
3.  char nazwisko[20];
4.  char imie [20];
5.  int rok studiow;
6.  char wydzial[20];
7.  };
8.  struct student studenci[3]=\   \\definicja i inicjalizacja tablicy struktur
9.  { {"Kowalski","Marcin",2, "EiA"},
10.  {"Iksinski","Andrzej",2,"EiA"},
11.  {"Malinowski","Dariusz",2,"EiA"} };
```

Powyższy fragment programu zawiera definicją typu strukturowego student oraz definicję i inicjalizację tablicy struktur studenci o trzech elementach, polom poszczególnych elementów tablicy nadawane są wartości podane w nawiasach, w kolejności takiej, w jakiej występują pola w definicji typu strukturowego

Przykład 101. *Program rekordy1 wczytujący i drukujący pojedyncze struktury*.

```
1.  #include<stdio.h>
2.  #include<string.h>
3.  int main(int argc, char **argv)
4.  {
5.  struct student
6.  {
7.  char nazwisko[20];
8.  char imie [20];
9.  int rokStudiow;
10.  char wydzial[20];
11.  };
12.
13.  struct student student1, student2;
14.  //Bezpośrednie nadawanie wartości polom struktur
15.  strcpy(student1.nazwisko,"Kowalski");
16.  strcpy(student1.imie,"Jan");
17.  student1.rokStudiow=2;
18.  strcpy(student1.wydzial,"EiA");// Do wczytywania pól struktury student 2 z klawiatury
19.  // zastosowano trzy różne funkcje wczytywania  pól typu tablica znaków, by pokazać różne sposoby
20.  // wczytywania takich pól
21.  printf ("\n Podaj dane 2-go studenta ");
```

```
22. printf ("\n    Nazwisko: ");
23. scanf("%s",student2.nazwisko);
24. printf ("\n          Imie: ");
25. fflush(stdin);
26. fgets(student2.imie,15,stdin);
27. printf("\n Rok studiow :");
28. scanf("%d",&student2.rokStudiow);
29. printf ("\n       Wydział: ");
30. scanf("%s",student2.wydzial);
31. /* Drukowanie  struktur student1 i student2 */
32. printf("\n\n Dane studenta 1 ");
33. printf("\n Nazwisko    :%s", student1.nazwisko);
34. printf("\n       Imie    :%s", student1.imie);
35. printf("\n Rok studiow:%d", student1.rokStudiow);
36. printf("\n     Wydzial  :%s", student1.wydzial);
37. printf("\n\n\n Dane studenta 2 ");
38. printf("\n Nazwisko    :%s", student2.nazwisko);
39. printf("\n       Imie    :%s", student2.imie);
40. printf("\n Rok studiow :%d", student2.rokStudiow);
41. printf("\n      Wydzial  :%s", student2.wydzial);
42. getch();
43. return 0;
44. }
```

Przykład 103. *Zapis struktur do pliku i odczyt.*

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  struct student
4.  {
5.  char nazwisko[20];
6.  char imie[20];
7.  };
8.
9.  int main(int argc, char argv[ ])
10.  {
11.  char ch;
12.  int i;
13.  FILE *fp;
14.  char s[20]="pliktest.dat";// definicja tablicy znakowej zawierającej nazwę pliku
15.   struct student tab[4]=\ // inicjalizacja tablicy struktur tab typu student
16.  {{"iksinski","jan"},{"ygrekowski","andrzej"},
17.  {"wojcik","marcin"},{"kowalski","marek"}};
18.
19.  struct student tab1[4],st;
```

```
20. fp=fopen(s,"w+b");
21.  fwrite(&tab,sizeof(struct student),4,fp);// zapis blokowy tablicy struktur do pliku
22. fseek(fp,0,SEEK_SET);// ustawienie wskaźnika pliku na początek pliku
23. i=1;
24. while (fread(&st,sizeof(struct student),1,fp))// odczyt z pliku zapisanych
25. {                                              // struktur i wyprowadzenie na ekran
26. printf("\n Dane studenta %d",i++);

27. printf("\n Nazwisko:%s", st.nazwisko);
28. printf("\n Imie:%s", st.imie);
29. }
30. getchar();
31. return 0;
32. }
```

Struktury zagnieżdżone

Typ strukturowy zagnieżdżony to taki typ, w których przynajmniej jedno z pól jest typu strukturowego.

```
struct nazwaTypu
      {
            typPola1   nazwaPola1;
            struct nazwaTypuStrukturowego  nazwaPola2;
            .
            .
            typPolaN  nazwaPolaN;
      };
```

Przykład 102. *Definicja typu strukturowego zagnieżdżonego.*

```
1.  struct data// definicja typu strukturowego data
2.  {
3.  int dzien;
4.  int miesiac;
5.  int rok;
6.  };
7.  struct daneStudenta// definicja typu strukturowego zagnieżdżonego
8.  {
9.  char nazwisko[20];
10.  char imie [20];
11.  struct data dataUr ;// pole struktury typu data
12.  int rokStudiow;
13.  char wydzial[20];
14.  };
15.  struct daneStudenta daneStudenta1, daneStudenta2;
```

Wprowadzanie/wyprowadzanie informacji z /do pól zmiennych strukturowych zagnieżdżonych

Sposób wprowadzania wartości do pól zmiennych strukturowych jest tutaj podobny jak dla pól struktur niezagnieżdżonych wymaga jednak użycia dodatkowych operatorów dostępu do pola struktury '.' (kropka). Schematycznie dla jednopoziomowego zagnieżdżenia można to przedstawić następująco:

nazwaZmiennejStrukturowej . NazwaPolaStrukturowego. NazwaPolaZagnieżdzonego,

gdzie nazwaZmiennejStrukturowej reprezentuje zmienną do której pola chcemy uzyskać dostęp, NazwaPolaStrukturowego jest nazwą pola typu strukturowego, a NazwaPolaZagnieżdzonego jest nazwą, do którego chcemy uzyskać dostęp. Zagadnienie to zostanie zilustrowane przykładem wprowadzania informacji do pola dataUr zmiennych strukturowych daneStudenta1 i daneStudenta2 typu daneStudenta zdefiniowanej powyżej.

1. Pola numeryczne

```
daneStudenta1. dataUr.dzien=2;          // przypisanie wartości 2 polu dzien
                                        //pola dataUr typu data

daneStudenta1. dataUr.miesiac=5;        // przypisanie wartości 5 polu miesiac
                                        //pola dataUr typu data

daneStudenta1. dataUr.rok=1990;         // przypisanie wartości 1990 polu rok
                                        //pola dataUr typu data


scanf("%d",&daneStudenta2.dataUr.dzien);        // wczytanie wartości z klawiatury i
                                                //zapis do pola dzien

scanf("%d",&daneStudenta2.dataUr.miesiac);      // wczytanie wartości z klawiatury i
                                                //zapis do pola miesiac

scanf("%d",&daneStudenta2.dataUr.rok)           // wczytanie wartości z klawiatury i
                                                //zapis do pola rok
```

2. Pola typu tablica znaków

Wprowadzanie informacji do pól zagnieżdżonych typu tablica znaków opiera się na tej samej zasadzie, wymaga jednak użycia albo jakiejś formy kopiowania łańcuchów, np. funkcji strcpy, a dla wprowadzania z klawiatury funkcji scanf z deskryptorem %s, gets czy też fgets.

Wyprowadzanie informacji odbywa się podobnie jak dla innych zmiennych.

```
printf ("\n   %d ", daneStudenta1. dataUr.dzien);
printf ("\n   %d ", daneStudenta1. dataUr.miesiac);
printf ("\n   %d ", daneStudenta1. dataUr.rok);
```

Inne własności typu strukturowego

Typ strukturowy może być również definiowany przy użyciu słowa typedef w sposób następujący

```
typedef struct
{
char nazwisko[20];
char imie[20];
} STUDENT;
```

STUDENT jest nowym typem strukturowym. Można wtedy definiować zmienne strukturowe typu STUDENT jako STUDENT student1, student2 ; Istnieje też możliwość definiowania zmiennych strukturowych bez podawania przedrostka (ang. structure tag).

```
struct
{
char nazwisko[20];
char imie[20];
} student3={"Nowak","Marcin"}, student4={"Smith","Martin"}, student5;
```

Inną operacją dozwoloną dla zmiennych strukturowych tego samego typu jest bezpośrednie kopiowane, np.

```
zmiennaStrukturowa1=zmiennaStrukturowa1;
```

Przykładowo, jeśli mamy dwie zmienne tego samego typu strukturowego można zrealizować następujące przypisanie

```
student5=student4;
```

Wskutek tego przypisania wszystkie pola zmiennej strukturowej student4 są kopiowane do pól zmiennej student5. Przy stosowaniu tego rodzaju kopiowania należy pod uwagę fakt, że w przypadku, gdy dany typ strukturowy zawiera pole typu wskaźnikowego, wskazujące na pewien obszar pamięci, to po skopiowaniu zmiennej strukturowej, pola wskaźnikowe w obu zmiennych strukturowych będą wskazywać na ten sam obszar pamięci.

Struktury i funkcje

Zmienne strukturowe (wartości typu strukturowego) mogą być parametrami funkcji jak również wartościami zwracanymi przez funkcje. Deklaracja takiej funkcji może mieć postać:

```
nazwaTypuStrukturowego  nazwaFunkcji( nazwaTypuStrukturowego parametrFormalny);
{
   ..........
return nazwaZmiennejStrukturowej;
```

}

Przykład 104. *Użycie funkcji zwracającej strukturę i funkcji pobierającej strukturę.*

```
1.  typedef struct      //definicja typu strukturowego STUDENT
2.  {
3.  char nazwisko[20];
4.  char imie[20];
5.  } STUDENT;
6.
7.  STUDENT wczytaj (void);  // deklaracja funkcji wczytującej zmienną strukturową
8.                                    //typu STUDENT i zwracającej wczytaną strukturę
9.  void drukuj (STUDENT st); // deklaracja funkcji drukującej zmienną
                        //strukturową
10.  int main(int argc, char* argv[])
11.  {
12.  int i;
13.  STUDENT studenci[4];    // definicja 4-elementowej tablicy struktur typu
                        //STUDENT
14.  for (i=0;i<4;i++)
15.  studenci[i]=wczytaj();      // wywołanie funkcji wczytaj i przypisanie struktury zwróconej
                                  //przez funkcję elementowi tablicy studenci
16.  for (i=0;i<4;i++)
17.   drukuj (studenci[i]);      // wywołanie funkcji drukującej tablicę, do funkcji przekazywane
18.                               // są kolejne elementy tablicy studenci
19.  getchar();
20.  return 0;
21.  }
22.
23.  STUDENT wczytaj( void)          // definicja funkcji wczytaj, funkcja nie ma
24.                                    //parametrów i zwraca strukturę typu STUDENT
25.  { STUDENT st;
26.  printf ("\  Nazwisko: ");
27.  scanf("%s",st.nazwisko);
28.  fflush(stdin);
29.  printf ("\n      Imie: ");
30.  scanf("%s",st.imie);
31.  fflush(stdin);
32.   return st;
33.  }
34.
35.  void drukuj( STUDENT st)         // funkcja drukuje strukturę typu STUDENT
36.  {
37.  printf("\n Nazwisko:%s", st.nazwisko);
38.  printf("\n Imie:%s", st.imie);
```

```
39. }
```

Wskaźniki do struktur

Dla typów strukturowych podobnie jak dla innych typów można definiować wskaźniki.

```
NazwaTypuStrukturowego  *nazwaZmiennejWskaznikowej;
```

Przykład 137. *Definicja wskaźnika do struktury i dynamiczna alokacja zmiennej strukturowej*.

```
STUDENT* pStudent;
pStudent=( STUDENT*) malloc(sizeof(STUDENT));     // alokacja pamięci dla
 //zmiennej strukturowej typu STUDENT, rzutowanie wskaźnika zwróconego przez
//funkcję malloc i przypisanie tego wskaźnika wskaźnikowi pStudent
```

Wskaźniki umożliwiają też utworzenie tzw. struktur samoreferencyjnych (ang. self-referential structures). Poniżej podano przykład definicji typu strukturowego studentRef, w którym występuje pole wskaźnikowe, będące wskaźnikiem do do struktury typu studentRef.

```
struct studentRef
{
char nazwisko[20];
char imie[20];
struct studentRef *next;    // pole typu wskaźnik do typu strukturowego studentRef
};
```

Taki typ strukturowy umożliwia tworzenie złożonych struktur danych takich jak listy i kolejki.

Operator ->

Aby uzyskiwać dostęp do pól struktury za pośrednictwem wskaźnika wprowadzono specjalny operator ->. Wynika to stąd, że operator dostępu do pola struktury " . " ma wyższy priorytet niż operator dereferencji " . ". Konstrukcja

```
*pStudent.nazwisko
```

oznacza nie dostęp do pola nazwisko, a dereferencję pola nazwisko traktowanego jako wskaźnik. Prawidłowy dostęp wymaga użycia nawiasów.

```
(*pStudent).nazwisko
```

Aby tego uniknąć wprowadzono operator -> umożliwiający stosowanie następującej konstrukcji

```
pStudent->nazwisko
```

16 Klasy pamięci, kwalifikatory i zasięgi

Specyfikatory klasy pamięci informują kompilator w jaki sposób przechowywać wartości poszczególnych zmiennych. Klasa pamięci określa typ pamięci, w której przechowywana jest zmienna, określa też moment utworzenia zmiennej, czas przechowywania jej wartości, jak również moment usunięcia danej zmiennej. Wyróżnia się następujące klasy pamięci:

-auto
-static
-register
-extern

Zmienna zdefiniowana wewnątrz funkcji otrzymuje domyślną klasę auto (zwana jest zmienną automatyczną). Definicję zmiennej automatycznej można zapisać następująco:

```
auto int liczba;
```

jednak zwykle słowo auto jest pomijane. Pamięć dla zmiennej jest alokowana (przydzielana) za każdym razem przed wejściem sterowania programu do bloku zawierającego definicję danej zmiennej i jest dealokowana (usuwana), gdy sterowanie programu opuszcza ten blok. Termin auto wywodzi się stąd, iż proces ten odbywa się automatycznie bez udziału programisty.

Zmienna typu auto jest widoczna tylko w bloku, w którym znajduje się jej definicja. Jeżeli zmienna typu auto jest jednocześnie definiowana i inicjalizowana, to jej inicjalizacja jest powtarzana za każdym razem, gdy sterowanie programu wchodzi do bloku ją zawierającego.

```
void fun ( int n, double x)
{
int i,j; // zmiennymi automatycznymi są  i, j

}
```

Klasa pamięci static

Zmienna o klasie pamięci static może być zdefiniowana wewnątrz bloku np. wewnątrz funkcji albo też na zewnątrz wszystkich bloków. Zmienne statyczne są tworzone przed rozpoczęciem wykonywania programu i istnieją przez cały czas jego wykonywania.

Definicja zmiennej statycznej typu int ma postać

```
static int liczba;
```

Zmienne statyczne zdefiniowane wewnątrz bloku mają ten sam zasięg co inne zmienne zdefiniowane w bloku. Jednak zachowują one swą wartość po opuszczeniu bloku przez sterowanie programu. Powoduje to, na przykład w przypadku funkcji, że wartość statycznej zmiennej lokalnej może być przechowywana pomiędzy kolejnymi wywołaniami funkcji.

Przykład 105. *Funkcja zliczająca i zwracająca ilość swoich wywołań*.

```
1. int fun (void)
2. {
3. static int liczba=0;          //definicja zmiennej statycznej liczba i inicjalizacja jej
4.                               //wartością 0.
5. int i=0;
6. i++;
7. liczba++;
8. .....
9. return liczba;
10. }
```

Zmienna liczba jest inicjalizowana tylko jeden raz przy pierwszym wywołaniu funkcji. Przy każdym kolejnym wywołaniu funkcji, wartość zmiennej liczba jest zwiększana o 1 i przechowywana między wywołaniami funkcji, gdy sterowanie programu opuszcza blok funkcji. Zmienna i jest zerowana przy każdym wywołaniu funkcji.

Zmienna statyczna zdefiniowana na zewnątrz funkcji

W przypadku zmiennej statycznej, która została zdefiniowana poza wszystkimi blokami (na zewnątrz wszystkich funkcji), definicja taka ogranicza widoczność zmiennej do pliku, w którym definicja ta wystąpiła.

Przykład 106. *Zasięg zmiennej statycznej*.

W skład projektu wchodzą dwa pliki : plik1.c i plik2.c

// plik1.c	// plik2.c
static double x; i int f1 **(void)** **{** . }	static int x; **char** * f2**(char** *s**)** **{** . .**}**

W obu plikach występują zmienne o nazwie x, jednak są one dostępne tylko tam, gdzie występują ich definicje.

Klasa pamięci register

Definiując zmienną typu register tworzy się zalecenie dla kompilatora, aby umieścił daną zmienną w rejestrze procesora, traktowanym jako komórka pamięci. Powodem stosowania takiego zalecenia jest czas dostępu, który jest zwykle dla rejestrów znacznie krótszy niż dla pamięci

zewnętrznej. Kompilatory mogą uwzględniać to zalecenie lub nie, z drugiej strony jednak sam kompilator stara się umieszczać w rejestrach częściej używane zmienne, np. liczniki pętli. Jeżeli zmienna nie stanie się zmienną register, to będzie zmienną automatyczną.

Przykład 107. *Użycie klasy pamięci register.*

```
1. #include <stdio.h>
2. int main()
3. {
4. register int i; // definicja zmiennej typu i klasy register
5. int sum=0;
6. for (i=1;i<1000;i++)
7. {
8. sum+=i;
9. printf("\n Suma=%d",sum);
10. getchar();
11. }
12. return 0;
13. }
```

Klasa pamięci extern (zmienne zewnętrzne)

Klasę pamięci extern mają tzw. zmienne zewnętrzne (globalne). Są to zmienne definiowane poza wszystkimi funkcjami w jednym z plików źródłowych składających się na program. Mogą być one dostępne w każdym miejscu programu. Po zdefiniowaniu są one dostępne w pliku źródłowym, poniżej ich definicji. Zmiennym tym w momencie definicji przypisuje się wartość 0. Jeśli chce się udostępnić zmienną w innych miejscach (powyżej definicji zmiennej w danym pliku lub w innych plikach źródłowych), należy użyć deklaracji zmiennej z użyciem słowa extern.

Przykład 108. *Użycie słowa extern.*

```
1. extern int x;// deklaracja zmiennej zewnętrznej x
2. void fun2(int)
3. int fun1( int a, int b)
4. {
5. int c;
6. fun2(x);// x jest zmienną zewnętrzną użytą jako argument
7. ..... // funkcji, konieczna jest wcześniejsza deklaracja
8. // zmiennej x przed jej użyciem w funkcji fun1
9. }
10. int x; // definicja zmiennej zewnętrznej x void fun2(int a)
11. {.....
12. }
```

Słowo extern określa, że obiekt o zewnętrznej kategorii konsolidacji jest zdefiniowany w innej części programu. Słowo extern może być też zastosowane w definicji. Deklaracja zmiennej staje się definicją, gdy zmienna jest inicjowana, tzn. nadaje się jej wartość początkową, np.

```
extern double y=5.0; // definicja zmiennej zewnętrznej y
```

Ważnym zastosowaniem specyfikatora extern jest sytuacja, gdy program składa z kilku plików, które mogą być osobno kompilowane i następnie konsolidowane (łączone). W sytuacji takiej trzeba przekazać do wszystkich plików informację o zmiennych zewnętrznych stosowanych w programie. Zmienne zewnętrzne można zdefiniować w jednym z plików, a w pozostałych plikach można umieścić deklaracje ze słowem extern.

```
// plik pierwszy
double x,y;
char ch;
int main( )
{/* */}
void fun1(void)
{
};
```

```
// plik drugi
extern double x,y;
extern char ch;
char * f2( char *s)
{ ...}
```

W pliku drugim zamieszczono deklaracje poszczególnych zmiennych zewnętrznych z pliku pierwszego z modyfikatorem extern. Informuje on kompilator, że deklarowane zmienne zostały zdefiniowane w innym miejscu programu, jednak deklaracja zawiera informacje o typach i nazwach zmiennych, ale kompilator nie musi rezerwować dla nich pamięci. Deklaracje extern umieszcza się zwykle w jednym pliku nagłówkowym, który dołącza się przy użyciu dyrektywy #include do wszystkich plików źródłowych.

Kwalifikatory typu

Każda zmienna oprócz nazwy, typu i klasy pamięci może też mieć kwalifikator typu. W języku C istnieją dwa kwalifikatory

-volatile

- const

Kwalifikator volatile (ulotny)

Kwalifikator ten wskazuje, że zmienna może być zmieniana spoza programu, w którym została zdefiniowana.

Przykład 109. *Zastosowanie kwalifikatora* volatile.

```
1. volatile int flagaOczekiwania=1;// definicja zmiennej ulotnej flagaOczekiwania i inicjalizacja
2.                                 //wartością 1
3. void czekaj (void);
4. int main ( )
5. {
6. czekaj();
7. }
8.
9. void czekaj (void)
10. {
11. while ( flagaOczekiwania) ;
12. }
```

Oczekiwanie będzie trwało, dopóki system operacyjny nie zmieni flagi oczekiwania, oczekiwanie to może być nieskończone. Kwalifikator volatile wskazuje, że dana zmienna może być zmieniona przez system operacyjny i kompilator nie powinien przeprowadzać optymalizacji związanych z tą zmienną (poprzez tworzenie kopii).

Kwalifikator const

Kwalifikator const jest kwalifikatorem służącym do definiowania specjalnych zmiennych, tzw. zmiennych ustalonych, których wartość, po jednokrotnym przypisaniu, nie może ulec zmianie.

Przykład 110. *Definicja zmiennej ustalonej (stałej).*

```
const int a=5;
```

W powyższej instrukcji a jest zmienną ustaloną (stałą). Zmiennym ustalonym przydzielana jest pamięć w przeciwieństwie do stałych tworzonych przy użyciu #define. Słowo const zmienia interpretację zdefiniowanego obiektu. Ogranicza ono możliwość użycia do odczytu, ale zachowuje informację o typie. Kwalifikator const może też służyć do ochrony przed zmianą parametrów funkcji przekazywanych przez zmienną.

Przykład 111. *Zastosowanie* const *do zabezpieczenia przed zmianą przekazywanego parametru tablicowego.*

```
1. void druk2D( int n, int m, const double x[n][m])// zastosowanie const dla
2. // parametru tablicowego x zabezpiecza tablicę przed zmianą wewnątrz funkcji
3. {
4.  int i,j;
```

```
5.  for (i=0;i<n;i++){
6.   for (j=0;j<m;j++)
7.    printf("%f  ",x[i][j]);
8.  printf("\n");
9.  }
10.  // x[0][0]=1; taka próba przypisania powoduje błąd kompilacji
11.  // i komunikat " assignment to read-only location", czyli próba
12.  // przypisania wartości dla lokacji tylko do odczytu
13.  }
```

Poniżej omówimy użycie kwalifikatora const dla wskaźników. Kwalifikator ten może być użyty w różny sposób, co ma określone konsekwencje odnośnie samego wskaźnika, jak i obiektów wskazywanych przez wskaźnik.

Zmienna wskaźnikowa zdefiniowana bez const w sposób następujący:

```
int *wsk1;
```

nie może przechowywać adresów stałych, natomiast może przechowywać adresy zmiennych i w tym przypadku można modyfikować zarówno samą zmienną wsk1 jak i wskazywaną komórkę *wsk1.

Zmienna wskaźnikowa zdefiniowana z const jako

```
const int *wsk2;
```

może przechowywać adresy stałych jak i zmiennych. Można zmieniać wartość wskaźnika, przypisując mu np. adres zmiennej x, wsk=&x, a następnie adres zmiennej y, natomiast nie można zmieniać komórki wskazywanej przez wskaźnik, czyli *wsk2.

Przy użyciu const można też utworzyć stały wskaźnik

```
const* int wsk3;
```

W tym przypadku można zmieniać wartość wskazywaną, czyli *wsk3, lecz nie można zmieniać samego wskaźnika wsk3.

Kwalifikator const może być też użyty jak niżej

```
const int const *wsk4;
```

Wskaźnik wsk4 jest wtedy stałym wskaźnikiem do stałej, nie można wtedy modyfikować ani wsk4 ani też *wsk4. Możliwe jest tylko zainicjowanie wskaźnika jakimś adresem.

Zasięgi w języku C

Zasięg, ściślej zasięg identyfikatora, oznacza te fragmenty programu, gdzie identyfikator może

być użyty. W języku C wyróżniamy następujące zasięgi

-zasięg bloku

-zasięg funkcji

-zasięg pliku

-zasięg prototypu

-zasięg programu

Zasięg bloku

Blok tworzą instrukcja złożona { } i przynajmniej jedna definicja użyta wewnątrz tej instrukcji.

Przykład 112. *Zasięg bloku.*

```
1. {// początek bloku zewnętrznego
2. double x;
3.
4. {
5. // blok wewnętrzny
6. int x[5];
7. // x jest tutaj tablicą , zmienna x z bloku zewnętrznego
8.  // jest przesłaniana przez x będący tutaj nazwą tablicy
9. .
10. }
11. // x jest zmienną typu double
12. }
```

W powyższym przykładzie występują dwie definicje zmiennej x, w bloku zewnętrznym x oznacza zmienną typu double, a w bloku wewnętrznym x jest nazwą tablicy. Po wyjściu sterowania programu z bloku wewnętrznego x ponownie oznacza zmienną typu double.

Zasięg funkcji

Zasięg funkcji mają parametry formalne i zmienne lokalne funkcji.

Przykład 113. *Zasięg funkcji.*

```
1. int oblicz( int x[], int n, int *pNp) // parametry x,n, *pNp, i,sumP i sumNp mają
2.                                       // zasięg funkcji
3. {
```

```
4.  int i; double sumP=0, sumNp=0;
5.  for (i=0;i<n;i++)
6.  {
7.  if (x[i]%2==0)
8.  sumP=sumP+x[i];
9.  else
10.  sumNp=sumNp+x[i];
11.  *pNp=sumNp;
12.  return xP;
13.  }
```

W przykładzie zarówno zmienne i, sumP, sumNp jak i parametry formalne x, n i pNp są zmiennymi lokalnymi funkcji, są dostępne tylko wewnątrz funkcji.

Zasięg pliku

Zasięg pliku mają wszystkie zmienne o klasie pamięci static zdefiniowane w danym pliku źródłowym poza wszystkimi funkcjami jak również funkcje o klasie pamięci static.

Zasięg prototypu

Zasięg prototypu jest zasięgiem identyfikatorów umieszczonych w deklaracji funkcji.

Przykład 114. *Zasięg prototypu*.

```
void wczytC99( int n, int m, int x[n][m]);
```

Powyższy wiersz jest prototypem funkcji. Zmienne *n*, *m*, *x* mają zasięg prototypu.

Zasięg programu

Zasięg programu mają funkcje oraz wszystkie identyfikatory zdefiniowane poza funkcjami nie posiadające klasy pamięci static.

Część II Programowanie strukturalne w języku C – Robert Smyk

1. Podstawy strukturalnego rozwiązywania złożonych problemów w języku C

1.1 Co to jest programowanie strukturalne?

Uznaje się, że termin *programowanie strukturalne* został wprowadzony przez Edsgera Dijkstra w roku 1969 podczas wygłoszonej przez niego prelekcji pt. „*Structural programming*" na konferencji o tematyce związanej z inżynierią oprogramowania. Główną ideą przyświecającą omawianej technice programowania jest dążenie do tego, aby program używał bloku o jednym wejściu i jednym wyjściu, jako podstawowej struktury kontrolnej. W ogólności, kod programu powinien rozpoczynać się tylko w jednym z góry ustalonym miejscu oraz kończyć się również w jednym ustalonym miejscu.

Należy zauważyć, że w okresie, kiedy termin programowanie strukturalne stawał się popularny, powstał język C. Omówiona wyżej właściwość podejścia strukturalnego ma odzwierciedlenie w szkielecie każdego programu pisanego w języku C. Program rozpoczyna wykonywanie od jednego znanego miejsca, którym jest pierwsza instrukcja zawarta w funkcji głównej `main()`. Zakończenie wykonywania programu jest także ustalone i następuje typowo po wykonaniu instrukcji `return 0`.

Opisywany sposób kodowania może być używany w wielu językach programowania, takich jak: asembler, Pascal, Fortran oraz wymieniony język C. Wszystkie te języki w sposób naturalny, poprzez wbudowane mechanizmy/komponenty dostępne dla programisty, wspierają strukturalny proces tworzenia kodu. Wyróżnia się trzy podstawowe komponenty charakterystyczne w procesie tworzenia kodu tą techniką: blok sekwencji instrukcji, blok wyboru i blok iteracji. Komponenty te niekiedy są nazywane *strukturami kontrolnymi*.

Blok sekwencji jest zbiorem instrukcji, które są wykonywane w ustalonym porządku: od pierwszej do ostatniej. Typowa sekwencja może zawierać deklaracje zmiennych oraz wywołania procedur lub funkcji. Pokażmy to na prostym przykładzie programu wyświetlającego resztę, którą powinien wypłacić kasjer osobie kupującej bilet:

```
1. int cenaBiletu,wplaconaGotowka;//instrukcja deklaracji zmiennych
2. cenaBiletu=7;//instrukcje przypisania
3. wplaconaGotowka=10;
4. printf("reszta do wydania:%d\n",wplaconaGotowka-cenaBiletu);//drukowanie informacji
   poprzez wywolanie funkcji bibliotecznej
```

Można łatwo zauważyć, że poszczególne instrukcje są wykonywane jedna po drugiej, bez zatrzymywania wykonywania programu. Należy także zwrócić uwagę na logikę ustalonego porządku wykonywania instrukcji: definicje zmiennych, przypisanie konkretnych wartości zmiennym, wykorzystanie zmiennych w funkcji drukującej komunikat na ekranie oraz obliczającej sumę. Istotne jest zachowanie tego a nie innego porządku wykonywania instrukcji. Przykładowo, gdybyśmy próbowali wykonać powyższe działania w innej kolejności, np. najpierw wywołali

procedurę wydruku, a dopiero później określili jaka jest cena biletu oraz ile za niego zapłacono, z oczywistych względów porządek sekwencji nie miałby sensu.

Blok wyboru, niekiedy nazywany selekcją (ang. selection), zapewnia kontrolę nad wykonywanym programem w zależności od aktualnego stanu programu, a precyzyjniej nad sekwencjami instrukcji. Typowym przykładem jest wykorzystanie konstrukcji *if-else* lub *switch-case*. Bez bloku wyboru niemożliwa jest realizacja zadań uzależnionych od aktualnie występujących warunków. Patrząc na wyżej zaprezentowany przykład widać, że gdyby próbowano zapłacić za bilet kwotą mniejszą niż wartość biletu, kasjer wydrukowałby paragon z resztą ujemną. Oczywiście taka sytuacja nie może mieć miejsca, więc program wymaga korekty:

```
1. int cenaBiletu,wplaconaGotowka;//instrukcja deklaracji zmiennych
2. cenaBiletu=7;//instrukcje przypisania
3. wplaconaGotowka=10;
4. if(wplaconaGotowka>cenaBiletu)
5.         printf("reszta do wydania:%d\n",wplaconaGotowka-cenaBiletu);//drukowanie
           informacji poprzez wywolanie funkcji bibliotecznej
```

Blok iteracji pozwala na wielokrotne wykonywanie sekwencji instrukcji. Typowy przykład to pętla na bazie konstrukcji *for* lub *while*. Do czego w prezentowanym przykładzie można by zastosować pętlę? Program mógłby drukować kilka kopii paragonu – do archiwum kasy oraz dla klienta. Wielokrotne wywołanie funkcji drukującej naturalnie wymagałoby zastosowania pętli.

Pierwotna teza głoszona przez twórców programowania strukturalnego mówi, że za pomocą trzech wyżej wymienionych struktur kontrolnych możliwe jest zrealizowanie dowolnej konstrukcji programowej. Można dowieść, że jest to prawda, chociaż niektóre języki programowania, np. asembler, a także język C, dopuszczają stosowanie instrukcji bezwarunkowego skoku, tzw. `goto`. Instrukcja tego typu pozwala na wykonanie skoku w dowolne miejsce przed lub za aktualnie wykonywaną instrukcją. Działanie takie zaburza ciągłość przebiegu wykonywania kodu, przede wszystkim wyklucza zasadę wykonywania programu "od instrukcji pierwszej do ostatniej". Wśród programistów panują opinie, że stosowanie `goto` jest niezgodne z ideą programowania strukturalnego.

Podstawowe założenia programowania strukturalnego można podsumować następująco:

- jedno wejście i jedno wyjście kodu modułu; główny kod programu powinien mieć początek i koniec w ustalonym miejscu (funkcja `main()` w języku C)
- kod tworzy się przy użyciu podstawowych konstrukcji: sekwencji instrukcji, selekcji, iteracji
- możliwe jest zagnieżdżanie jednych konstrukcji w innych, np.: użycie sekwencji instrukcji w bloku selekcji
- należy ograniczyć do minimum stosowanie skoku bezwarunkowego (najlepiej wyeliminować)
- program powinien składać się z niedużych jednostek nazywanych procedurami; dobrym praktycznym zwyczajem jest budowanie procedur, które nie zawierają mniejszych podprogramów

Obecnie synonimem programowania strukturalnego stało się programowanie proceduralne. Niektórzy traktują je także jako rozszerzenie koncepcji programowania strukturalnego. Podstawową

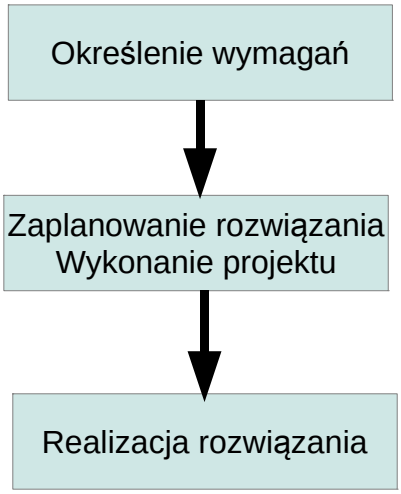
jego cechą jest realizacja pewnych bloków funkcjonalnych programu w postaci procedur. W języku C procedury realizuje się w postaci funkcji nie zwracających wartości, stąd termin procedura jest często zamiennie stosowany z terminem funkcja. Funkcja może być traktowana jako blok instrukcji. Do funkcji można przekazywać dane poprzez jej parametry. Zmienne deklarowane w ramach funkcji mają charakter lokalny. Typowo zmienne tworzone w ramach funkcji istnieją od momentu przekazania sterowania do funkcji – podczas jej wywołania w programie – do momentu opuszczenia funkcji. Taki mechanizm zapewnia podstawowy poziom separacji danych, zapewniając, że zmiany zachodzące w procedurze/funkcji nie będą miały wpływu na inne bloki programu.

1.2 Realizacja zadań przy użyciu programowania strukturalnego

Dosyć oczywiste wydaję się być stwierdzenie, że oprogramowanie tworzy się aby sprostać konkretnemu zapotrzebowaniu. Można przy takim podejściu pominąć wszelkie przypadki szczególne, uwzględniające np. tworzenie programów podczas nauczania danego języka programowania czy też rozwijanie własnych zainteresowań. W typowym przypadku trzeba rozwiązać jakiś konkretny problem, czy też wykonać konkretne zadanie.

Chcąc sprostać wyzwaniu realizacji dużego zadania, można spróbować podzielić je na kilka mniejszych składowych, pozostając w optymistycznym przekonaniu, że mniejsze zadania uda się łatwiej rozwiązać. Tego typu podejście wydaje się być jak najbardziej naturalne, gdy następnym krokiem realizacji rozwiązania jest już tylko napisanie odpowiedniego programu. Zwykle jest to jeden z ostatnich etapów.

W podejściu inżynierskim podczas rozwiązywania bardziej złożonych zadań wykonuje się przynajmniej trzy podstawowe etapy:



Rys 1. Planowanie rozwiązania złożonego problemu

Proces określania wymagań obejmuje sformułowanie problemu, który program powinien rozwiązać. Określenie wymagań wiąże się również z nakreśleniem bardzo ogólnej wizji rozwiązania. Rozróżnia się zwykle dwa rodzaje wymagań: funkcjonalne i pozafunkcjonalne. Wymagania funkcjonalne opisują funkcje lub funkcjonalności, które ma realizować program. W odniesieniu do zasad programowania strukturalnego zazwyczaj wymagania funkcjonalne mają

odwzorowanie w kodzie programu w postaci funkcji/procedur lub zbiorów funkcji/procedur. Wymagania pozafunkcjonalne dotyczą zazwyczaj wydajności lub niezawodności działania programu.

Zaplanowanie rozwiązania lub wykonanie projektu programu polega w gruncie rzeczy na zaplanowaniu kształtu kodu programu. Na tym etapie uwzględnia się wymagania funkcjonalne i wyodrębnia się stosowne do nich procedury, które należy zrealizować w celu ukształtowania rozwiązania docelowego – przyjmującego wówczas formę programu głównego. Procedury wyodrębnione na podstawie wymagań funkcjonalnych mogą nadal być zbyt złożone aby można je było w stosunkowo prosty sposób zrealizować w postaci pojedynczych funkcji. Wówczas należy szukać możliwości wyodrębnienia mniejszych, bardziej szczegółowych bloków, które będzie można zrealizować w postaci funkcji.

Omawiane podejście do projektowania jest w literaturze znane pod pojęciem *modularności*. Idealnym sposobem budowania oprogramowania byłoby zebranie istniejących komponentów i ich zestawienie. Modularność jest cechą, która umożliwia taki sposób działania. Należy jednak pamiętać o tym, że komponenty – moduły powinny być ze sobą kompatybilne, gdyż tylko to zapewni możliwość ich użycia w jednym programie. Modularność niesie także korzyści na etapie tworzenia kodu. Małe moduły zwykle można napisać szybciej – niższym nakładem pracy. Krótszy kod łatwej się testuje, co wpływa na efektywność pracy programisty. Równie istotną cechą jest możliwość wykorzystania uniwersalnych modułów (np. procedur sortujących lub wyszukujących) w innych programach. Należy wspomnieć, że chociaż cechę modularności można przypisać w opisywanym tu kontekście do podejścia proceduralnego, jest ona uznawana za typową właściwość programowania zorientowanego obiektowo.

Na etapie planowania rozwiązania zapada również decyzja o wyborze języka programowania. Wybór ten wymaga zachowania ostrożności oraz weryfikacji możliwość realizacji założonych zadań w wybranym języku programowania. Przykładowo język C umożliwia realizację zadania o praktycznie dowolnej złożoności. Istnieją jednak przykłady pokazujące, że nie zawsze opłaca się stosować rozwiązania bazujące na C (np. aplikację bazującą na graficznym interfejsie użytkownika we współczesnych systemach MS Windows nieporównywalnie łatwiej jest wykonać przy użyciu tzw. Windows Forms /obiektowa wersja biblioteki programisty MS Windows/ w C# niż w Windows API /strukturalna wersja biblioteki programisty MS Windows/ w C).

Ostatni z wymienionych etapów tworzenia rozbudowanego projektu programistycznego to realizacja rozwiązania. Punkt ten obejmuje głównie proces tworzenia kodu, uruchamianie oraz weryfikację. W wielu źródłach dotyczących inżynierii oprogramowania podkreśla się, że etap weryfikacji jest nieodzownym składnikiem wszystkich trzech przedstawionych punktów realizacji rozwiązania złożonych projektów.

Jeżeli problem, który mamy zamiar rozwiązać pisząc program został dobrze zrozumiany, a jego rozwiązanie zostało szczegółowo zaplanowane, sprawność tworzenia kodu zależy głównie od umiejętności posługiwania się składnią danego języka programowania. Oczywiście, jeżeli zadanie jest złożone i wymaga realizacji dziesiątek funkcji, proces ten może być czasochłonny. Należy jednak mieć na uwadze, że niepoprawne wykonanie wcześniejszych czynności projektowych lub całkowite ich zignorowanie w najbardziej optymistycznym wariancie doprowadzi do kodu, którego uruchomienie może kosztować dużo wysiłku. Ponadto nieudolnie zaimplementowany kod będzie zapewne stanowił poważną przeszkodę w przypadku chęci poprawy wydajności aplikacji, chęci rozwoju lub przy weryfikacji błędów w logice funkcjonowania programu.

Podsumowując powyższe bardzo ogólne rozważania, można przyjąć, że przy realizacji projektu programistycznego istotne jest:

- rozbicie zadania na mniejsze zadania, które można zrealizować równolegle lub niezależnie;

każde mniejsze zadanie implementuje się w programie w postaci funkcji
- wyodrębnienie zależności między podzadaniami i przedstawienie ogólnego rozwiązania w postaci np. diagramu procedur/funkcji; wyodrębnienie zależności pozwoli ustalić parametry funkcji (czyli zmienne) oraz wartości przez nie zwracane
- zastosowanie trzech podstawowych etapów inżynierii oprogramowania: określenie wymagań, zaplanowanie rozwiązania zadania, przystąpienie do realizacji rozwiązania (kodowanie); punkt ten jest istotny ze względu na powodzenie realizacji projektu.

1.3 Obszary stosowania programowania strukturalnego w języku C

Język C jest zaliczany do grupy języków strukturalnych wysokiego poziomu oraz ogólnego przeznaczenia. Według normy ISO/IEC 9899-1999 zawiera on 37 słów kluczowych i nieco ponad 20 typów podstawowych. Sam język C nie oferuje zbyt wiele. Jednak w powiązaniu z podstawowymi bibliotekami funkcji, wachlarz zastosowań języka C staje praktycznie nieograniczony.

Język C zapewnia wysoką efektywność programu. Okazuje się, że program napisany w tym języku może być w szczególnych przypadkach równie szybki jak program napisany w języku asemblera. Z tego też powodu język C był pierwotnie wykorzystywany do realizacji oprogramowania systemowego lub samych systemów operacyjnych. Warto zauważyć, że jądro (jest to główny składnik systemu operacyjnego) systemów Linux/Unix napisane jest właśnie w tym języku. W systemach firmy Microsoft, cała podstawowa biblioteka programisty tzw. Windows API została napisana w języku C.

Istnieje wiele języków wyższego poziomu niż C, które są jednak zakorzenione w C. Pierwszym rozszerzeniem C był język C++. Podstawową funkcjonalnością języka C++ odróżniającą go od C jest wdrożenie wspomagania dla programowania zorientowanego obiektowo. Innymi współcześnie używanymi językami, dla których pierwotna inspiracją był C są bardzo popularne języki takie jak: PHP, Java oraz C#.

Profesjonalista będzie raczej kierował się zasadą dostosowania wyboru języka programowania w zależności od rozwiązywanego zadania. Współcześnie stosowane języki programowania są bardzo uniwersalne. Jednak przykładowo, systemy operacyjne lepiej jest realizować w języku nie nakładającym ograniczeń oraz efektywnym takim jak C czy C++. Niedługie programy prototypowe lub wprawki programistyczne często pisze się w językach skryptowych takich jak Perl lub Python. Graficzne interfejsy użytkownika aplikacji uruchamianych lokalnie szczególnie wygodnie realizuje się w języku C# platformy .NET czy języku Java. Nowoczesne aplikacje internetowe pisze się w PHP w połączeniu z Flash lub innymi.

W typowych zadaniach inżynierskich język C może znaleźć zastosowanie jako:

- standard programowania systemów wbudowanych (aplikacje sieciowe, systemy czasu rzeczywistego np. ISIX-RTOS, przetwarzanie sygnałów, układy automatyki itp.)
- standard programowania mikrokontrolerów od serii Intel 8051, poprzez popularne Atmel AVR, do bardzo zaawansowanych mikrokontrolerów 32-bitowych z rdzeniem ARM Cortex M; język C posiada bogatą kolekcję operatorów co zapewnia możliwość efektywnego wykonywania obliczeń niskiego poziomu, np. maskowanie czy przesuwanie
- standard programowania systemowego tzw. soft procesorów, np. Xilinx MicroBlaze lub Altera Nios II; istnieją gotowe biblioteki pozwalające na realizację typowych zadań stawianych współczesnym systemom wbudowanym

- do realizacji oprogramowania systemowego w Unix/Linux
- standard programowania z wykorzystaniem Windows API
- baza do realizacji projektów niewymagających GUI (graficznego interfejsu użytkownika) w MS Windows lub Unix/Linux; przykładem mogą być tu biblioteki funkcji użytkownika lub biblioteki systemowe, sterowniki urządzeń

Ze względu na konieczność zapewnienia dużej wydajności jaką oferuje C jest on również często stosowany do realizacji oprogramowania rozrywkowego, czyli gier komputerowych.

Język C charakteryzuje się pewnymi specyficznymi właściwościami, które mogą mieć wpływ na obszary jego zastosowania. C pozwala programistom na większą niż inne języki programowania kontrolę nad sposobem zapisu danych i ich inicjalizacją. Istotny wpływ na efektywność C ma to, że kod jest uruchamiany bez zaawansowanych mechanizmów kontroli bezpieczeństwa, jako tzw. kod niezarządzany. Przykładowo podczas dostępu do tablic przez indeksy lub wskaźniki nie jest kontrolowana poprawność tych indeksów. Takie działanie ma oczywiście wpływ na wydajność programu, ale wymaga od programisty zwiększonego poziomu dyscypliny np. przy realizacji funkcji operujących na tablicach.

Mniej zaawansowany czytelnik, może odczuwać potrzebę jaskrawego zilustrowania technik programowania strukturalnego przy użyciu C. Za przykład może posłużyć rozwinięcie programu obsługi kasy biletowej z punktu 1.1. Wykonując skrócony proces określenia wymagań projektowych można przyjąć, że obsługa kasy biletowej wymaga:

- określenia rodzaju biletu
- archiwizowania informacji o operacjach pieniężnych
- koordynacji procedury zapłaty oraz wydawania reszty
- wydruku paragonu

Powyższe zadania są sformułowane bardzo ogólnie, wymagają zatem podzielenia na podzadania. Pierwsze wymaganie ma charakter użytkowy. Można zauważyć, że program będzie wymagał wdrożenia odpowiedniego interfejsu użytkownika, pozwalającego na łatwy wybór rodzaju biletu. Funkcja wyboru biletu oraz inne funkcje mogą być podfunkcjami interfejsu użytkownika. Zadanie archiwizowania informacji o operacjach pieniężnych może wymagać funkcji: zapisu danych na dysk, odczytu danych z dysku oraz wyszukiwania danych.

Funkcje koordynacji zapłaty i wydawania reszty oraz wydruk paragonu są ze sobą powiązane. Informacje zawarte na paragonie są ściśle zależne od rodzaju biletu, oraz sposobu zapłaty. Najczęściej zdarza się, że klient płaci określonym nominałem niekoniecznie pokrywającym się z ceną biletu – tu zachodzi konieczność obliczenia reszty oraz odnotowania tej operacji na paragonie. Funkcje koordynujące zapłatę i wydawanie reszty będą w rzeczywistości zawierały procedurę odblokowania szuflady na pieniądze, tak aby kasjer miał dostęp do gotówki. W omawianym przypadku zostanie to pominięte.

Przedstawiony zestaw czynności związanych z obsługą kasy odzwierciedla plan funkcjonowania systemu. Ważne jest przy tym ustalenie odpowiedniej kolejności wykonywania poszczególnych funkcji. Można nanieść prezentowany tok rozumowania na schemat blokowy, celem weryfikacji poprawności założeń oraz ustalenia kolejności wykonywania procedur (Rys 2).

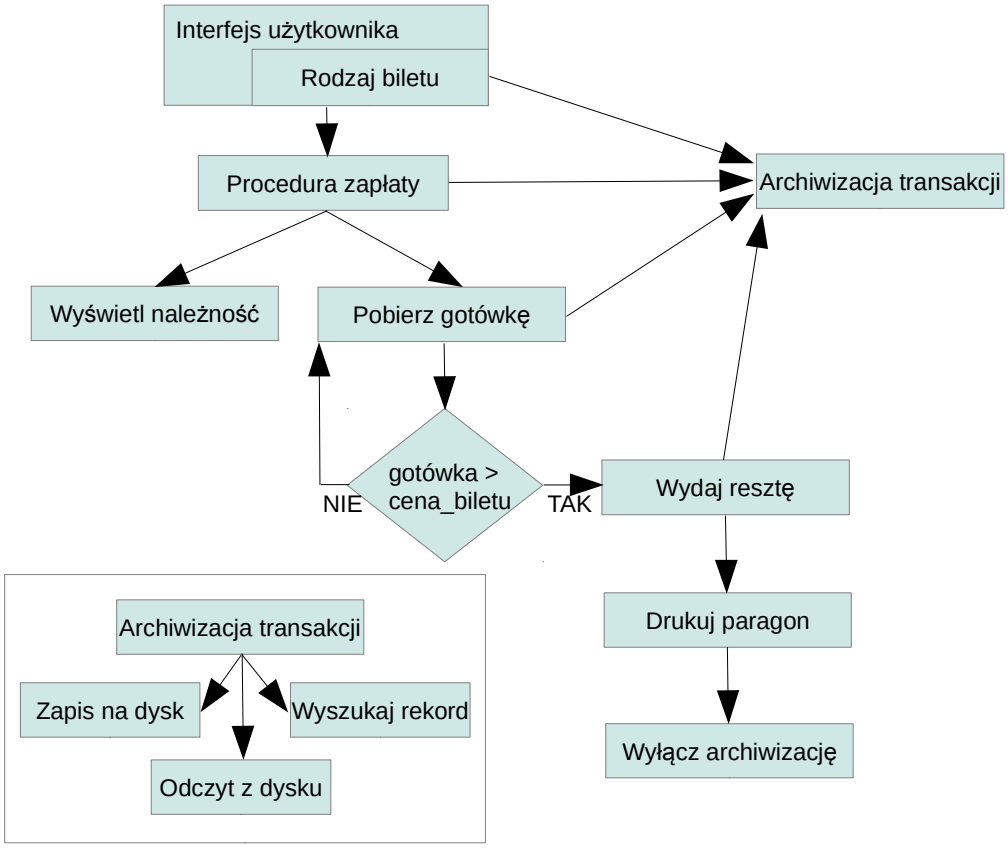
Na podstawie przedstawionego planu można spróbować skonstruować szkielet programu głównego wraz z funkcjami. Funkcja główna `main()` będzie zawierała w tym przypadku tylko jedną funkcję.

Pseudokod C programu do zarządzania kasą biletową

```
void zaplata( double cena_biletu)
{     double gotowka;
      wyswietl_naleznosc();
      gotowka = pobierz_gotowke();

      if(gotowka>cena_biletu)
            wydaj_reszte();
      else
            gotowka = pobierz_gotowke();
            //wykonuj w sekwencji nieskonczonej lub alternatywnie: rezygnuj_z_transakcji();
}

void interfejs_uzytkownika()
{
      while(1)
      {
            achiwizuj_wlacz();
            rodzaj_biletu();
            zaplata(7);
            drukuj_paragon();
            archiwizuj_wylacz();
      }
}

int main()
{
      interfejs_uzytkownika();
      return 0;
}
```

Wykonanie programu rozpoczyna się od funkcji interfejs_uzytkownika(). Dalej w trybie pętli nieskończonej wykonywane są w ustalonej kolejności funkcje zawarte w definicji funkcji interfejsu użytkownika. Zauważmy, że zaplata() zawiera zestaw funkcji realizujących proces od pobrania gotówki do wydania reszty – czyli zakończenia transakcji.

Zaprezentowany pseudokod C został napisany z wykorzystaniem zasad programowania strukturalnego, które przedstawiono w punkcie 1.1 oraz 1.2 niniejszego rozdziału. Należy zauważyć, że w programie dla uproszczenia pominięto definicję funkcji archiwizacji transakcji. To zadanie pozostawia się czytelnikowi. Autor proponuje również wykonanie samodzielnej analizy przedstawionego problemu, szczególnie porównanie schematu blokowego z kodem programu. Warto także pokusić się o samodzielne zaplanowanie rozwiązania i wykonanie podobnego do powyższego szkieletu rozwiązania.



Rys 2. Schemat funkcjonowania przykładowego systemu obsługi kasy biletowej

1.4 Ograniczenia programowania strukturalnego

Podstawowe zalety programowania strukturalnego w odniesieniu do innych technik to:

- proste główne zasady - korzystny wpływ na naukę programowania w tym stylu
- mało skomplikowane problemy generują prosty kod
- możliwość grupowania kodu w funkcje pozwalające na użycie go w wielu miejscach bez konieczności powtarzania
- dość łatwy sposób śledzenia przebiegu programu
- modularna struktura kodu
- duża efektywność programu wykonywalnego
- powszechnie dostępne kompilatory oraz prosty sposób ich użycia

Podstawowe wady programowania strukturalnego mają znaczenie szczególnie przy realizacji dużych projektów programistycznych. Należą do nich:

- brak technik ochrony danych; zmienne są tworzone w obrębie funkcji lub w programie głównym
- brak możliwości realizacji kodu odwzorowującego rzeczywiste obiekty; co prawda istnieje możliwość tworzenia funkcjonalności (funkcje) ale brak jest prostych mechanizmów

kojarzenia z nimi właściwości (zmiennych/danych) jak to ma miejsce w językach obiektowych (klasa jako zbiór cech i funkcjonalności obiektu)
- mniej elastyczne niż w podejściu obiektowym możliwości tworzenia nowych typów danych
- w świecie realnym wiele problemów dotyczy przetwarzania różnych struktur danych; w programowaniu strukturalnym pierwszoplanowy jest opis funkcjonalności (funkcje), a nie właściwości (dane)

Spośród porównań występujących w literaturze najczęściej zestawia się programowanie strukturalne i obiektowe. Bardziej szczegółowe zasady programowania obiektowego przy użyciu C++ zostaną poruszone w części III tekstu. Na tym etapie można jednak spróbować odnaleźć elementy, które na określonym poziomie abstrakcji są dla obu technik równoważne:

Programowanie zorientowane obiektowo	Programowanie strukturalne
Atrybut	Zmienna
Metoda	Funkcja
Komunikacja za pomocą tzw. wiadomości (messages)	Wywołanie
Obiekt	Moduł

Warto wspomnieć, że programowanie proceduralne jest w pewnym sensie elementem programowania obiektowego. Na niskim poziomie, funkcje – metody zawierają kod, który jest zwykle tworzony w postaci monolitycznego bloku, składającego się z typowych struktur kontrolnych, o których była mowa na początku niniejszej części.

2. Programowanie strukturalne w języku C

2.1 Wskazówki praktyczne

Wśród bardziej doświadczonych programistów panuje przekonanie, że bardzo istotne jest pisanie kodu w określonym stylu. Styl to sposób pisania kodu, który będzie czytelny zarówno dla autora jak i dla innych programistów. Kod powinien być zrozumiały i przejrzysty. Można wymienić inne cechy takie jak: prosta logika, konwencjonalne używanie języka, naturalne wyrażenia, nazwy skojarzone ze znaczeniem, odpowiedni układ, komentarze. Powinno unikać się raczej niespotykanych konstrukcji lub skrajnych uproszczeń, choć te drugie język C naturalnie umożliwia.

Praca w zespole programistów wymaga stosowania jednolitego sposobu kodowania. Pozwala on na łatwiejsze czytanie mojego kodu przez innych oraz innym - mojego kodu. Pewne konwencje mogą być narzucone z góry, jednak zawsze zasadą jest przestrzeganie ogólnie przyjętych zasad programowania w danym języku.

W dalszej części tego punktu zostaną przytoczone elementarne cechy ogólnie przyjętych konwencji programowania w C. Na najniższym poziomie języki C oraz C++ są identyczne. Dlatego też większość przytoczonych tu wskazówek można stosować do programowania w C++ oraz

innych językach pochodnych.

Nazewnictwo

Nazwa zmiennej lub funkcji określa dany obiekt oraz przekazuje pewien zasób informacji o jego przeznaczeniu. Nadając nazwę powinno się pamiętać się o zwięzłości, ale jednocześnie łatwości zapamiętania, również łatwości wymówienia oraz zawarciu w niej opisu jakiejś cechy nazywanego obiektu. Przy nazewnictwie stosuje się pewną ogólną zasadę, która mówi: `używaj nazw opisowych dla definicji globalnych, a nazw krótkich dla definicji lokalnych.`

Zmienne globalne oraz funkcje powinny mieć nazwy opisowe sugerujące ich przeznaczenie i sposób użycia w programie. Dobrze widziane jest przy tym, opatrzenie takich składników programu stosownymi komentarzami. Przykładowo, deklarując zmienną globalną określającą liczbę komórek tablicy, w której zawarto pomiary temperatury można by ją zadeklarować w sposób następujący:

```
int lpomt; //liczba pomiarow temperatury
```

Nazwa zmiennej jest niezbyt czytelna, lepiej byłoby ją nazwać:

```
int liczbaPomiarowTemperatury; //liczba pomiarow temperatury
```

W przypadku zmiennych lokalnych naturalne jest stosowanie nazw krótkich. Przykładem mogą być tu zmienne sterujące pętlą nazywane często 'i' lub 'j'.

Dobrą praktyką jest skojarzenie nazw funkcji zgodnie z czynnością, którą wykonuje funkcja. Można przyjąć za zasadę, że nazwa funkcji jest zawsze dwuczłonowa: *czasownikRzeczownik*. Przykładowo funkcję zapisującą dane do tablicy można nazwać: `zapiszTablicę`. Inny przykład, funkcja sortująca po wybranym polu `Imie` rekordu zdefiniowanego w postaci struktury

```
struct dane{
    char Imie[30];
    char Nazwisko[40];
    int nrindeksu;
}
```

może nazywać się `sortujDaneImie`.

Stosowanie nazw dla wartości liczbowych

Liczba, która nagle pojawia się w programie, bez informacji o jej znaczeniu zmniejsza czytelność kodu. Takie liczby są czasem nazywane liczbami magicznymi. Nadając im nazwy możemy spowodować, że kod będzie bardziej zrozumiały. Dodatkowo, jeżeli dana wartość występuje częściej, pozbędziemy się konieczności odszukiwania jej, gdy zajdzie konieczność zmiany. Wystarczy zmienić jej wartość w miejscu przypisania.

Istnieją również przypadki, w których nie opłaca stosować się nazw. Niektóre funkcje biblioteczne wymagają podawania rozmiaru obiektu w bajtach. Posługując się operatorem `sizeof` można łatwo wyznaczyć rozmiar dowolnej zmiennej:

```
char bufor[256];
fgets(bufor, sizeof(bufor), stdin);//bufor - cel zapisu danych, stdin - zrodlo pobierania danych (klawiatura)
```

W bardziej złożonym w programie korzystającym z tablic, często będzie zachodziła potrzeba odwołania się do rozmiaru tablicy. Można zapamiętać tę wartość w zmiennej, jednak gdy tablica zmieni rozmiar lub typ, trzeba będzie nanieść korektę ręcznie. Niestety język C nie posiada wbudowanej funkcji określającej rozmiar tablicy. Dla dowolnej tablicy sprawdzi się następująca makroinstrukcja:

```
//sizeof(tablica) – calkowity rozmiar tablicy w bajtach
//sizeof(tablica[0]) – rozmiar w bajtach pojedynczej komorki tablicy
#define ROZMTAB(tablica) (sizeof(tablica) / sizeof(tablica[0]))
```

którą można używać w każdej instrukcji wymagającej podania rozmiaru tablicy, np.:

```
int liczby[80],i;
for(i=0;i<ROZMTAB(liczby);i++)
/*dalsza czesc kodu*/
```

Wyrażenia i instrukcje

Konstruując wyrażenia i instrukcje zawsze warto dbać o utrzymywanie kodu w takiej postaci, aby jego kształt pozwalał jednoznacznie zrozumieć działanie programu. Należy starać się pisać kod jak najbardziej przejrzysty dla danego zadania. Najbardziej oczywistą regułą jest stosowanie jednolitego stylu wcięć oraz układu nawiasów klamrowych. Wiele współczesnych edytorów posiada automatyczny mechanizm utrzymywania kodu w ogólnie przyjętym porządku. Przykładowo, zbyt zwięźle i nieczytelnie sformatowany kod może wyglądać następująco:

```
for(i=0;i<15;znaki[i++]=' ');
```

Lepsza jego wersja :

```
for(i=0;i<15;i++)
{
    znaki[i]=' ';
}
```

Zastosowanie nawiasów klamrowych obejmujących jedną instrukcję, którą steruje pętla może zostać uznane przez doświadczonego programistę za przesadę. W takiej postaci przykład jednak jest bardziej zrozumiały i dosłownie jednym spojrzeniem określa się jego działanie.

Używanie wyrażeń w ich naturalnej postaci wpływa również pozytywnie na zwiększenie czytelności. Jako przykład niejasnej konstrukcji wyrażeń może posłużyć instrukcja warunkowa:

```
if(!(a<=b) && !(b>a))
```

którą z powodzeniem można zastąpić instrukcją o identycznym działaniu, ale czytelną:

```
if((a>b) && (b<=a))
```

Przykład ilustruje także inną zalecaną zasadę, stosowanie nawiasów w celu rozwiania niejasności. Nawiasy określają kolejność wykonywania działań. Doświadczony programista uzna,

że nawiasy w tym przypadku są zbędne. Będzie po prostu wiedział, że operatory relacji mają wyższy priorytet niż operatory logiczne. W przypadku stosowania operatorów, które nie są związane ze sobą zaleca się używanie nawiasów. W języku C oraz innych pochodnych istnieje ryzyko popełnienia błędu związanego z nieznajomością zasad pierwszeństwa operatorów.

Jednolity styl kodu

Początkujący programiści często borykają się z problemem nieuporządkowanego kodu. Nieprzewidywalny układ programu bardzo utrudnia zrozumienie sensu programu. Przykładowo chaos może powstawać, gdy w celu odczytania czy zapisania elementów tablicy stosujemy raz pętlę *for* innym razem *while* a przy tym bez potrzeby indeks zmieniamy raz od pierwszej pozycji do ostatniej innym razem od ostatniej do pierwszej. Gdy przedstawione działania są za każdym razem wykonywane w taki sam sposób, wówczas każda zmiana sugeruje różnicę, na którą należy zwrócić uwagę.

Z jednolitym sposobem kodowania związana jest umiejętność stosowania tzw. idiomów programistycznych – zwyczajowych sposobów pisania powtarzających się i typowych fragmentów kodu, które są przyjęte przez doświadczonych programistów. Można nawet uznać, że istotną podstawową zasadą nauki języka programowania jest oswajanie się z jego idiomami.

Przykładem powszechnego idiomu może być postać pętli. Zastanówmy się nad realizacja kodu przypisującego do kolejnych komórek tablicy wartości 0. Postawione zadanie można rozwiązać w C na wiele sposobów:

```
i=n;
for(;i>=0;n--)
      tablica[i]=0;
```

lub

```
for(i=0;i<n;)
      tablica[i++]=0;
```

Rozwiązania te są poprawne, jednak idiomatyczna postać pętli znana z literatury oraz praktyki jest następująca:

```
for(i=0;i<n;i++)
      tablica[i]=0;
```

Makroinstrukcje w roli funkcji

W języku C możliwa jest realizacja zadań, typowo przypisywanych funkcjom w postaci tzw. makroinstrukcji. Makroinstrukcja pozwala uniknąć kosztów dodatkowych, które są związane z wywołaniem funkcji. Spotyka się zatem opinie, że makroinstrukcja zapewnia wyższą wydajność programu niż funkcja, co uznawane jest za zaletę.

Makrodefinicje działają na zasadzie podstawienia tekstowego. Argumenty formalne makrodefinicji zastępowane są przez argumenty wywołania. Dalej w programie jest umieszczany pełny tekst makroinstrukcji zamiast jej wywołania.

Jednym z problemów, który występuje przy stosowaniu makroinstrukcji w roli funkcji jest to, że argument, który występuje wiele razy w makrodefinicji, może być wielokrotnie obliczany.

Spróbujmy przeanalizować kod makra `isupper()`, które może pochodzić z pliku nagłówkowego `<ctype.h>`:

```
#define isupper(c) ((c) >= 'A' && (c) <='Z')
```

W treści powyższego makra argument formalny `c` występuje dwukrotnie. Użycie powyższej makrodefinicji w instrukcji `while(isupper(c=getchar()))` będzie powodowało odrzucenie pobranego znaku, jeżeli okaże się większy niż 'A' oraz pobranie następnego znaku i porównanie ze znakiem 'Z'. Zauważmy, że poprawnie napisana procedura powinna wykonać test tego samego znaku, przyrównując go do 'A' i jednocześnie do 'Z'.

Komentarze

Umiejętne stosowanie komentarzy ułatwia czytanie programu. Nie powinno dostarczać się w nich informacji trywialnych. Poniższy przykład pokazuje **błędne** stosowanie komentarzy, typowe dla niedoświadczonych programistów:

```
int i=0; //zadeklarowalem 'i' i zainicjowalem wartoscia 0
int N=10//zadeklarowalem 'N' i zainicjowalem wartoscia 10
while(i<N)//poczatek petli
{
      i++;//zwiekszylem 'i' o 1
}//koniec petli...
```

Komentarze powinny zawierać tylko informacje, które dotyczą własności realizowanego algorytmu lub np. gromadzą w jednym miejscu informacje rozproszone w programie źródłowym. Gdy kod realizuje skomplikowany algorytm warto dodać do niego komentarz wskazujący nazwę algorytmu, lub źródło pochodzenia. Komentarz poprzedzający funkcję ułatwia jej zrozumienie:

```
/* max: przekazuje liczbe calkowita, a lub b, o maksymalnej wartosci */
int max(int a, int b)
{
      return a>b ? a : b;
}
```

Przy stosowaniu komentarzy warto również pamiętać o kilku dodatkowych regułach:

- nie stosuj komentarzy do złego kodu, lepiej staraj się go poprawić; jeżeli komentarz przeważa nad kodem, jest to oznaką, że prawdopodobnie kod trzeba poprawić.
- nie stosuj komentarzy sprzecznych z kodem; zdarza się, że podczas rozbudowywania programu, czy usuwania błędów zapomina się o zmianie czy dostosowaniu komentarzy
- staraj się pisać możliwie czytelny kod; jeżeli np. nazwiemy zmienne i funkcje zgodnie z ich przeznaczeniem nie trzeba będzie ich komentować – dobry kod wymaga mniej komentarzy

2.2 Cel wykorzystania funkcji

Umieszczanie całego kodu programu w funkcji głównej `main()` może być uzasadnione tylko w przypadku, gdy program nie przekracza kilkunastu lub kilkudziesięciu wierszy kodu. W wielu sytuacjach nawet krótkie programy pisze się implementując główne bloki kodu w postaci funkcji użytkownika. Takie podejście ma przynajmniej trzy podstawowe zalety. Po pierwsze, logika tak tworzonego programu będzie prostsza, ponieważ w jego części głównej wywoływane są tylko funkcje, realizujące konkretne zadanie. Po drugie, funkcje mogą być łatwo przenoszone do innych programów. Ostatnia zaleta jest związana z dystrybucją zadań w zespole, gdy oprogramowanie pisane jest przez zespół programistów. Rozwój kodu podzielonego na funkcje może być wówczas łatwiej koordynowany.

Istotę wykorzystania funkcji wyjaśnimy na przykładzie. Spróbujmy zastanowić się nad rozwiązaniem zadania znalezienia liczb pierwszych z określonego z góry zakresu. Zastosujemy w tym celu znany algorytm – *sito Eratostenesa*. Polega on z grubsza na wielokrotnym przeszukiwaniu tablicy wypełnionej początkowo wartościami *prawda (true)*, a w miarę wykonywania algorytmu – również wartościami *fałsz (false)*. Precyzyjniej, należy wykonywac następujące czynności:

1. zadeklaruj tablicę N wartości logicznych
2. wypełnij tablicę wartościami *true*
3. przeprowadź analizę tablicy
    1. ustaw pozycję początkową na element o indeksie 2
    2. liczba 2 jest pierwsza, więc wszystkie jej wielokrotności nie są liczbami pierwszymi
    3. oblicz wielokrotności liczby 2 i przypisz wartości *false* do elementów tablicy o indeksach pokrywających się z wielokrotnościami 2
    4. przejdź na początek tablicy (indeks 2)
    5. znajdź pierwszy na prawo element o wartości *true*
    6. element ten jest liczbą pierwszą, wprowadź wartości *false* do elementów tablicy o indeksach pokrywających się z wielokrotnością indeksu bieżącego elementu o wartości *true*
    7. proces analizy tablicy wykonuj dopóki nie dojdziesz do jej końca, realizując punkt 5

Program realizujący omawiany tu algorytm może mieć następującą postać:

```
1. #include <stdio.h>
2.  #include <stdbool.h>//C99: obsluga typu bool
3.
4.  int main()
5.  {
6.      const int N=100;//zakres szukania liczb pierwszych
7.      bool tabLog[N];//tablica wartosci logicznych
8.      int poz, i;//poz - poczatek szukania l. pierwszych
9.
10.
11.      for(i=0;i<N;i++)
12.          tabLog[i]=true; //wypelnij tablice wartosciami true
13.
14.
```

```
15.     /* Analiza tablicy */
16.     for(poz=2;poz < N;poz++)
17.     {
18.        if(tabLog[poz])//jezeli element = true to
19.        {
20.           for(i=2;i*poz < N;i++)//okreslam wielokrotnosci jego indeksu
21.           {
22.              tabLog[i*poz]=false;//i eliminuje elementy,
23.                                  //ktorych indeksy nie sa liczbami pierwszymi
24.           }
25.        }
26.
27.     }
28.
29.     poz=2;
30.     for(i=poz;i<N;i++)
31.       if(tabLog[i]) printf("%d\t",i); //drukuj liczby pierwsze
32.     getchar();
33.     return 0;
34. }
```

Przykład ilustruje rozwiązanie zadania w postaci zwartego bloku programu. Zauważmy, że program staje się dla nas czytelny tylko z tego powodu, że jesteśmy zaznajomieni z przedstawionym wyżej opisem problemu. Jeżeli autor programu przedstawiłby jedynie surowy kod, mało który programista – nawet bardziej doświadczony – w krótkim czasie zdołałby sklasyfikować problem, którego rozwiązaniem jest program. Zapewne można by poczynić odpowiednie kroki w celu zawarcia opisu działania programu np. w postaci komentarza. Warto jednak spróbować zastanowić się czy istnieje możliwość wykonania pewnych zmian w strukturze kodu, które spowodowały by jednoznaczne dopasowanie rozwiązania – programu do problemu, bez stosowania wylewnych komentarzy.

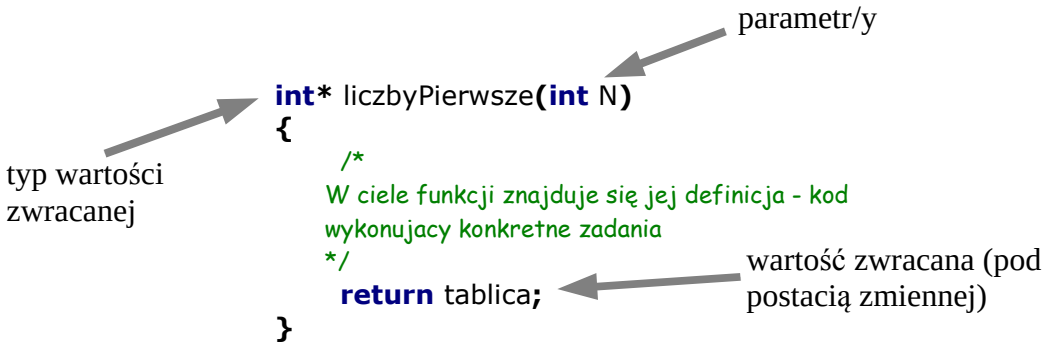
Odnosząc się do pierwszego akapitu niniejszego punktu, uważny czytelnik zaproponuje wyodrębnienie funkcji, która realizuje główną część algorytmu. Jeżeli dalej programista zastosuje odpowiednią nazwę dla tej funkcji, np.: `znajdzLiczbyPierwsze()`, każdy znający podstawy języka C będzie potrafił bez zastanowienia powiedzieć do czego służy program. Wyodrębnienie funkcji spowoduje również redukcję kodu programu głównego w `main()`, programista będzie też posiadał gotowe rozwiązanie, które można będzie wykorzystać w innym programie. Potwierdzają się zatem korzyści wymienione na początku tego punktu.

2.2.1 Podstawowe zasady użycia funkcji

Podczas tworzenia funkcji należy zdefiniować cztery składniki:

- typ wartości zwracanej przez funkcję
- nazwę funkcji
- listę parametrów

- ciało funkcji



Dobrym nawykiem przed wykonaniem wywołania funkcji, jest jej wcześniejsze zadeklarowanie. Deklaracja funkcji, niekiedy nazywana prototypem, pozwala kompilatorowi na ocenę poprawności wywołania funkcji. Kompilator zwraca uwagę m.in. na liczbę parametrów oraz zgodność ich typów z deklaracją. Gdy kompilator napotka na wywołanie funkcji, wygeneruje odpowiedni kod przekazujący wartości argumentów, wywołujący funkcję oraz - jeżeli funkcja zwraca wartość – odczytujący wartość przez nią zwróconą. W przypadku gdy informacja na temat funkcji nie zostanie przekazana w postaci deklaracji, kompilator przyjmuje dwa główne założenia: funkcja jest wywołana z odpowiednią dla niej liczbą argumentów oraz funkcja zwraca wartość `int`. Pierwsze założenie prowadzi do błędów, gdy programista wywołał funkcję z nieodpowiednią liczbą parametrów. Drugie założenie prowadzi do błędów, gdy funkcja zwraca inne wartości niż `int`. Spróbujmy prześledzić działanie poniższego programu:

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     double pi;
6.
7.     pi = liczbaPI();
8.     printf("%lf",pi);
9.     getchar();
10.    return 0;
11. }
12.
13. double liczbaPI()
14. {
15.    return 3.14;
16. }
```

Na pierwszy rzut oka nie ma w nim błędów. Funkcja `liczbaPI()` zwraca liczbę 3.14 typu `double`. W programie zadeklarowano zmienną `pi`, która ma taki sam typ jak wartość zwracana przez funkcję `liczbaPI()`. Dlaczego jednak ten program się nie skompiluje? Kompilator GCC informuje o następującym błędzie dotyczącym instrukcji `pi = liczbaPI();` :

```
error: conflicting types for 'liczbaPI'
```

Jaki jest powód konfliktu typów? Brak deklaracji funkcji `liczbaPI()` spowodował, że kompilator przyjął domyślnie typ `int` wartości zwracanej przez funkcję.

Prototypy funkcji bibliotecznych języka C zapisane są w plikach nagłówkowych. Jest to najbardziej właściwe miejsce umieszczenia prototypów. Dołączenie plików nagłówkowych do kodu źródłowego, np. `#include <stdio.h>`, umożliwia użycie funkcji bibliotecznej w programie. Prototypy funkcji, które zdefiniowane są w tym samym pliku co główny program, umieszcza się przed funkcja główną `main`.

W języku C dopuszczalne jest definiowanie funkcji bez prototypu. Definicję standardowo umieszcza się przed programem głównym zawartym w `main()`. Taka praktyka jest jednak krytykowana przez doświadczonych programistów, gdyż utrudnia stworzenie biblioteki funkcji. Nieważne czy zamierzamy w przyszłości rozwijać program tworząc bibliotekę czy nie, warto stosować prototypy.

Powracając do przykładu programu realizującego *sito Eratostenesa* spróbujmy zastanowić się jak wyodrębnić w nim fragment, który można zrealizować jako funkcję `liczbyPierwsze()`. W programie tak naprawdę jedyną wielkością ustalaną przy znajdywaniu liczb pierwszych jest zmienna N – górna granica obszaru poszukiwań. Można zauważyć, że ta zmienna będzie dobrym kandydatem nadającym się na parametr funkcji. Czy funkcja ta powinna zwracać wartości? Zależy czy będzie używana tylko w kontekście programu z przykładu – wówczas może tylko drukować liczby pierwsze. Chcąc zapewnić funkcji `liczbyPierwsze()` uniwersalne użycie, czytaj - stworzyć uniwersalny interfejs, powinniśmy zadbać aby zwracała tablicę liczb pierwszych. Aby w programie można było wydrukować tablicę liczb pierwszych potrzebna jest informacja o tym ile liczb pierwszych zostało znalezione. Wielkość ta jest znana dopiero po wykonaniu funkcji. Należy zatem rozważyć sposób przekazania tej wartości przez funkcję. Mając na uwadze jednolitość interfejsu, lepiej napisać funkcję w taki sposób, aby wymagane obiekty nie były zwracane przez funkcję a przekazywane przez parametry funkcji. Zadanie to może być jednak trudne dla początkującego, ze względu na ewentualną konieczność przekazywania do funkcji wskaźnika do tablicy liczb pierwszych. Pamiętajmy również o tym, że funkcja może zwracać tylko jeden obiekt określonego typu, a my chcemy przekazać przynajmniej dwa.

W poniższym rozwiązaniu zdecydowano się na funkcję zwracającą wskaźnik do tablicy liczb pierwszych. Funkcja pobiera górną granicę obszaru poszukiwania liczb pierwszych, oraz przekazuje poprzez parametr `iLiczb` rozmiar tablicy liczb pierwszych. Analizę rozwiązania pozostawia się czytelnikowi, jako ćwiczenie. W tym celu warto zacząć od skompilowania programu i jego uruchomienia.

```
1.  #include <stdio.h>
2.   #include <stdbool.h>//C99: obsluga typu bool
3.  #include <stdlib.h>
4.
5.  /* Opis algorytmu: http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes */
6.
7.  /* sitoEratostenesa: przekazuje liczbyPierwsze jako tablice liczb pierwszych z zakresu do N;
8.  iLiczb - ilosc znalezionych liczb pierwszych */
9.   int* sitoEratostenesa(const int N, int *iLiczb)
10.  {
11.      int *liczbyPierwsze;
12.      bool tabLog[N];//tablica wartosci logicznych obszaru poszukiwania l. pierwszych; std C99
13.      int poz, i, j;//poz - poczatek szukania l. pierwszych
```

```
14.
15.     for(i=0;i<N;i++)
16.        tabLog[i]=true; //wypelnij tablice wartosciami true
17.
18.     /* Analiza tablicy */
19.     for(poz=2;poz < N;poz++)
20.     {
21.        if(tabLog[poz])//jezeli element = true to
22.        {
23.           for(i=2;i*poz < N;i++)//okreslam wielokrotnosci jego indeksu
24.           {
25.              tabLog[i*poz]=false;//i eliminuje elementy,
26.                                  //ktorych indeksy nie sa liczbami pierwszymi
27.           }
28.        }
29.     }
30.
31.     /* ile znaleziono liczb pierwszych? */
32.     *iLiczb=0;
33.     poz=2;
34.     for(i=poz;i<N;i++)
35.       if(tabLog[i]) (*iLiczb)++;
36.     /*tworzona dynamicznie tablica do przechowania wszystkich znalezionych liczb pierwszych*/
37.     liczbyPierwsze = (int*)malloc((*iLiczb)*sizeof(int));
38.
39.     /*przepisz wszystkie znalezione liczby pierwsze*/
40.     j=0;
41.     for(i=poz;i<N;i++)
42.       if(tabLog[i]) liczbyPierwsze[j++]=i;
43.
44.       return liczbyPierwsze;
45. }
46.
47. /* Program  sito Eratostenesa */
48.  int main()
49. {
50.     const int N=100;//zakres szukania liczb pierwszych
51.     int *primes=NULL;//do uzycia w funkcji sitoEratostenesa w roli tablicy liczb pierwszych
52.     int iliczb, i;//iliczb - ilosc liczb pierwszych
53.
54.     /* Znajdz liczby pierwsze z zakresu 0...N */
55.     primes = sitoEratostenesa(N, &iliczb);
56.
57.     for(i=0;i<iliczb;i++)
58.        printf("%d\t",primes[i]);
59.     getchar();
60.     return 0;
61. }
```

2.3 Realizacja projektu jednoplikowego

Programy wykorzystujące tylko kilka funkcji użytkownika można realizować jako tzw. projekty jedno plikowe. Zaletą takiego podejścia jest archiwizacja kodu całego programu łącznie z funkcjami użytkownika w jednym miejscu. Podstawową wadą – brak możliwości wykorzystania funkcji użytkownika w innym programie bez konieczności kopiowania ich kodów w inne miejsce docelowe.

Rozwiązania jedno plikowe można realizować dla mniejszych zadań, projektów – dla których nie planuje się rozwoju w przyszłości lub w przypadku wczesnego stadium programowania zadania. Niekiedy zdarza się, że testując program mamy potrzebę bieżącego weryfikowania kodów funkcji. Wtedy również może sprawdzić się pomysł ulokowania kodów funkcji oraz programu głównego w jednym pliku.

Jeżeli nie potrafimy przewidzieć kierunku rozwoju tworzonego programu, a przy tym decydujemy się na realizację rozwiązania w postaci projektu jednoplikowego, warto kierować się kilkoma zasadami. Pomogą one w łatwiejszej reorganizacji kodu w przypadku np. konieczności utworzenia biblioteki funkcji.

Przypomnijmy na początek, jak przekazywane są do funkcji parametry. Wszystkie argumenty w języku C są przekazywane przez wartość. Oznacza to, że funkcja otrzymuje kopie wartości argumentów. Zauważmy jednak, że jeżeli do funkcji zostanie przekazana tablica, modyfikowanie jej elementów w funkcji skutkuje zmianą wartości elementów tablicy w programie głównym. Funkcja poprzez technikę zwaną wywołaniem przez referencję, odwołuje się do tej samej tablicy co wywołujący funkcję program.

Powróćmy teraz do przykładu ostatnio przytoczonej wersji programu *sito Eratostenesa*, w której zastosowano specjalizowaną funkcję. Przy realizacji funkcji zaszła potrzeba przekazania rozmiaru tablicy zawierającej liczby pierwsze. Informacja ta jest istotna przy odwoływaniu się do tablicy, np. w przypadku drukowania wartości jej elementów. Istnieją przynajmniej dwa sposoby przekazania tej informacji: umieszczenie jej w zmiennej globalnej do której mają dostęp wszystkie funkcje (czytaj – mogą ją modyfikować), lub zastosowanie tzw. zmiennej statycznej `static`, która nie traci wartości po zakończeniu wykonywania funkcji. Dlaczego nie skorzystano z tych możliwości?

Zmienna globalna lub inaczej zewnętrzna jeżeli jest zadeklarowana w tym samym pliku, jest ogólnie dostępna dla wszystkich funkcji zdefiniowanych w danym miejscu. Można było zatem uprościć sposób wywołania funkcji podając do niej „zwykłą” zmienną globalną deklarując rozmiar tablicy liczb pierwszych `iLiczba` nie jako wskaźnik, ale jako zmienna globalna. Dostęp do niej byłby możliwy zarówno z funkcji `sitoEratostenesa()` oraz programu głównego `main()`. Faktem jest, że gdy funkcja `sitoEratostenesa`() obliczyłaby rozmiar tablicy liczb pierwszych to w programie głównym wielkość ta była by również dostępna. Spowodowali byśmy jednak uzależnienie programu głównego od zmiennej zewnętrznej, zadeklarowanej poza nim. Budujemy dzięki temu mniej zrozumiały kod.

Najbardziej istotny jest jednak fakt, że gdyby zaszła potrzeba wykorzystania naszej funkcji w innym programie, należałoby pamiętać o przeniesieniu definicji funkcji oraz skojarzonej z nią zmiennej globalnej. W przypadku jednej zmiennej powiązanej z funkcją, aby pamiętać o omawianym skojarzeniu, można umieścić jej deklarację w sąsiedztwie deklaracji/definicji funkcji. Jednak i tak może to prowadzić do potencjalnych błędów interpretacji znaczenia powiązania zmiennej z funkcją. Jeżeli nie zachodzi wyraźna potrzeba stosowania zmiennych globalnych w

omawianym kontekście, powinno się z nich rezygnować.

Spróbujmy krótko zastanowić się nad użyciem zmiennych `static`. Gdy wymienione słowo zostanie użyte w definicji funkcji, lub deklaracji zmiennej znajdującej się poza blokiem, to funkcje i zmienne są dostępne tylko z pliku źródłowego. Jeżeli zostanie użyte przy deklaracji zmiennej wewnątrz bloku kodu, to zmienia się klasa zapisu zmiennej z automatycznej na statyczną. Zasięg zmiennej nie zmienia się. Powoduje to, że raz utworzone zmienne klasy `static` istnieją przez cały czas wykonywania programu. Przykładowo, nie są one niszczone gdy program wychodzi z funkcji i ponownie do niej wraca.

Odnosząc się do programu z przykładu, można by zadeklarować wewnątrz funkcji `sitoEratostenesa()` zmienną `static int iLiczba,` i wiedząc o tym, korzystać z niej w programie głównym. Wymagało by to jednak umieszczenia dodatkowego komentarza opisującego sposób przekazania tej informacji przez naszą funkcję. Takie działanie można porównać do produkcji urządzenia elektronicznego, np. telewizora, w którym włącznik zasilania jest tak ukryty, że w celu jego odnalezienia należy przeczytać całą instrukcję użytkownika. Podsumowując rozważania na temat użycia w omawianym kontekście zmiennej statycznej, tego typu podejście staje się oczywiście niepraktyczne.

Na koniec, warto zwrócić uwagę na sposób rozmieszczenia kodu źródłowego w projekcie jednoplikowym. Kod programu może być planowany według prostego szablonu:

- przed programem głównym umieść deklaracje funkcji użytkownika
- dalej umieść funkcję `main()` zawierającą program główny
- poniżej programu głównego umieść definicje funkcji

Przykład właściwego planowania układu kodu w projekcie jendnoplikowym

```
1. /*pliki naglowkowe*/
2. #include <stdio.h>
3.
4. /*makroinstrukcje*/
5. #DEFINE GRANSZUKANIA 100
6.
7. /*deklaracje zmiennych globalnych*/
8. //...
9.
10. /*deklaracje funkcji ućytkownika*/
11. int* sitoEratostenesa(const int N, int *iLiczb);
12. void drukujTabliceInt(int *tab, int N);
13.
14. /*prorgam glowny*/
15. int main()
16. {
17.     /*kod prorgamu glownego*/
18.     return 0;
19. }
20. /*definicje funkcji uzytkownika*/
21. int* sitoEratostenesa(const int N, int *iLiczb)
22. {
23.       /*kod zrodlowy
24.     funkcji sitoEratostenesa()*/
25. }
```

```
26. void drukujTabliceInt(int *tab, int N)
27. {
28.       /*kod zrodlowy
29.     funkcji drukujTabliceInt()*/
30. }
```

Jako ćwiczenie warto wykonać modyfikację przykładu programu *sito Eratostelesa*, dostosowując kod źródłowy do przedstawionej wyżej konwencji.

2.4 Realizacja projektu wieloplikowego

Praktycznie większość oprogramowania tworzonego w C, i innych podobnych językach , realizuje się w postaci tzw. projektów wieloplikowych. Projektem wieloplikowym można nazwać program, który zawiera definicje funkcji poza plikiem kodu programu głównego. Stosując takie rozplanowanie kodu programu można łatwo uporządkować definicje funkcji w zależności od ich przeznaczenia, przyjmując odpowiednie nazewnictwo pliku. Przykładowo kody funkcji realizujących różnego rodzaju algorytmy sortujące można umieścić w pliku *sortowanie_int.c*. Nazwa pliku informuje o jego zawartości. Nie musimy przeglądać jego zawartości, żeby zorientować się, że prawdopodobnie zawiera funkcje sortujące tablice elementów typu `int`.

W projektach wieloplikowych wykorzystuje się zazwyczaj następujący sposób planowania fragmentacji kodu:

- istnieje jeden plik o nazwie *main.c* lub *program_glowny.c* lub innej kojarzącej się z miejscem osadzenia programu głównego
- plik z programem głównym zawiera dyrektywy dołączenia plików nagłówkowych zawierających prototypy funkcji użytkownika, które są wykorzystywane w programie; dzięki temu kompilator posiada informacje o deklaracjach funkcji i może przygotować się na prawidłowy sposób ich wykorzystania (omówione w punkcie 2.2.1); nie dołącza się plików źródłowych dyrektywą include
- definicje funkcji pogrupowane tematycznie, w zależności od przeznaczenia, przechowywane są w oddzielnych plikach; każdy plik może zawierać definicje rodziny funkcji o podobnym przeznaczeniu, np. sortujące – *sortowanie.c*, wyszukujące – *wyszukiwanie.c* itd.; kod zawiera dyrektywę dołączenia pliku nagłówkowego zawierającego deklaracje funkcji
- deklaracje funkcji, a krócej - ich prototypy, przechowywane są w plikach nagłówkowych; są one pogrupowane tematycznie a ich nazwy odpowiadają nazwom plików źródłowych. Pliki nagłówkowe w języku C rozróżniane są poprzez rozszerzenie *.h*, np.: *sortowanie.h*, *wyszukiwanie.h* itd.

Drugoplanową zaletą omawianej realizacji projektu wieloplikowego jest to, że zawarte deklaracje w plikach nagłówkowych stanowią nieformalne źródło informacji o sposobie użycia tych funkcji. Doświadczony programista potrafi określić sposób użycia funkcji na podstawie oceny znajomości typu wartości zwracanej przez daną funkcję oraz typów parametrów. Zakładając, że funkcje nazwane są stosownie do przeznaczenia, na podstawie analizy zawartości pliku nagłówkowego jesteśmy w stanie określić poprawny sposób użycia funkcji i to bez czytania komentarzy.

Prezentowany sposób planowania dystrybucji kodu jest charakterystyczny dla tzw. zintegrowanych środowisk programistycznych IDE (ang. Integrated Development Environment)

wspomagających proces tworzenia, uruchamiania i konserwacji oprogramowania. Pozwala on na grupowanie funkcji danej rodziny w biblioteki funkcji. Współczesne kompilatory udostępniają opcję kompilowania kodu składającego się tylko z definicji funkcji oraz ich deklaracji do pliku wykonawczego, mającego charakter biblioteki. W tym przypadku biblioteka nie zawiera programu głównego a tylko zestaw funkcji. Proces tworzenia biblioteki użytkownika zostanie poruszony dalej.

Pliki biblioteczne w systemach MS Windows mają rozszerzenie *.dll*. Mogą one być niezależnie dołączane lub kojarzone z dowolnym programem wykorzystującym funkcje z danej biblioteki. Przykładowo, w katalogu `C:\Windows\System32` można odnaleźć pokaźny zbiór standardowo instalowanych bibliotek systemowych lub skojarzonych z zainstalowanym w systemie operacyjnym oprogramowaniem. Istnieje cały zbiór bibliotek systemowych z których standardowo korzysta większość programów uruchamianych pod kontrola MS Windows. Tworzenie i stosowanie oddzielnych bibliotek zapewnia większą modułowość oprogramowania. Wydzielając część funkcji „na zewnątrz" jesteśmy w stanie radykalnie zmniejszyć rozmiar pliku uruchomieniowego programu (w MS Windows pliki z rozszerzeniami .exe). Warto zauważyć, że prawie każda komercyjna aplikacja posiada plik uruchomieniowy o rozmiarze nie przekraczającym kilkuset kB czy kilku MB, przy czym rozmiar typowego dużego programu może sięgać nawet kilku GB.

Powróćmy do przykładu programu *sito Eratostenesa*, próbując zrealizować nasz projekt w konwencji wieloplikowej. W punkcie 2.3 zdołaliśmy wyodrębnić w programie funkcję `sitoEratostenesa`(). Mając na uwadze wyżej przedstawione wskazówki, można przenieść definicję funkcji do pliku źródłowego o nazwie *liczby_pierwsze.c* a deklarację do pliku *liczby_pierwsze.h*. Program główny można umieścić w pliku *program_glowny.c*. Całość będzie zorganizowana następująco:

Zawartość pliku *prorgam_glowny.c*:

```
1. /*dolaczenie naglowkow bibliotecznych*/
2. #include <stdio.h>
3. #include <stdbool.h>//C99: obsluga typu bool
4. #include <stdlib.h>
5. /*dolaczenie naglowka lokalnego*/
6. #include "liczby_pierwsze.h"
7.
8.
9. int main()
10. {
11.     /*kod prorgamu glownego*/
12.     return 0;
13. }
```

Zawartość pliku *liczby_pierwsze.h*

```
1. #ifndef _LICZBY_PIERWSZE_H_
2. #define _LICZBY_PIERWSZE_H_
3.
4. int* sitoErastotenesa(const int N, int *iLiczb);
5.
6. #endif
```

Zawartość pliku *liczby_pierwsze.c*

```
1. #include <stdlib.h>
2. #include <stdbool.h>
3. #include "liczby_pierwsze.h"
4.
5. /* sitoEratostenesa: ...*/
6.  int* sitoEratostenesa(const int N, int *iLiczb)
7. {
8.       /*kod zrodlowy funkcji*/
9. }
```

Przedstawiony przykład wymaga kilku słów komentarza. Dyrektywa `#include` powoduje skompilowanie zawartości dołączanego pliku w taki sposób, jakby jego kod znajdował się w miejscu dołączenia. Sposób obsługi tej dyrektywy jest następujący: preprocesor usuwa dyrektywę i wstawia zawartość danego pliku. Takie działanie ma pewne negatywne konsekwencje. W przypadku dołączenia danego pliku nagłówkowego w wielu innych plikach, które dalej są dołączane do programu głównego może dojść do wielokrotnego dołączenia tego pliku do źródła programu głównego. Istnieje proste rozwiązanie przedstawionego problemu. Stosuje się funkcję kompilacji warunkowej, zawartą w pliku *liczby_pierwsze.h.* Dzięki kompilacji warunkowej plik ten będzie dołączany pierwszą napotkaną dyrektywą `#include`. Niektóre kompilatory nie dopuszczają do kompilacji projektu wieloplikowego, w którym występuje wielokrotnie dyrektywa dołączenia tego samego pliku. Środowiska zintegrowane takie jak Visual Studio udostępniają standardową możliwość tworzenia pliku nagłówkowego opatrzonego dyrektywą kompilacji warunkowej.

2.5 Wykorzystanie funkcji bibliotecznych

Przy realizacji złożonych projektów dobrą praktyką jest, jak wcześniej wspomniano, organizowanie funkcji w oddzielnych plikach. Siłą języka C jest bogactwo gotowych funkcji, które zorganizowane są w biblioteki. Zawierają one zestawy funkcji zorganizowanych w rodziny o ustalonym przeznaczeniu.

Twórcy języka C zaopatrzyli go w pokaźny zbiór bibliotek dostępnych wewnętrznie, z poziomu kompilatora. Funkcje zorganizowane w postaci bibliotek nazywane są funkcjami bibliotecznymi. Wykorzystujemy je w prawie każdym programie. Przykładowo, wyświetlając napis za pomocą instrukcji `printf(''Menu glowne\n'');` wywołujemy funkcję `printf()` zdefiniowaną pierwotnie w pliku *stdio.c*, dla którego prototypy zapisane są w *stdio.h*. Ogólnie wiadomo, że dołączając do programu dyrektywę `#include <stdio.h>` mamy możliwość korzystania z wewnętrznych funkcji bibliotecznych *stdio*. Podczas kompilacji program linkujący współpracujący z używanym przez nas kompilatorem zadba o to, żeby skompilowany kod biblioteki *stdio* został połączony z plikiem wykonawczym (*.exe) naszego programu.

Spróbujmy przyjrzeć się nieco bliżej procesowi tworzenia oraz użycia własnej biblioteki funkcji. Przykład ten ilustruje zasadę tworzenia biblioteki zewnętrznej (użytkownika). Przyjmujemy, że w jednym katalogu znajdują się pliki wykorzystywane w przykładzie z punktu 2.4: *program_glowny.c, liczby_pierwsze.c* oraz *liczby_pierwsze.h*. Najpierw plik *liczby_pierwsze.c*

należy skompilować to postaci tzw. object code – kod pośredni. Następnie skompilować go do postaci *liczby_pierwsze.dll*. W przypadku tworzenia biblioteki funkcji na którą składa się wiele plików źródłowych, lepiej wykorzystać jedno z dostępnych IDE. Bardziej dydaktyczny w naszym przypadku będzie sposób utworzenia własnej biblioteki .dll, zawierającej funkcję `sitoErastotenesa()`, zademonstrowany przy użyciu wywołań kompilatora MinGW (wersja kompilatora GCC dla systemów MS Windows) z wiersza poleceń. W celu uzyskania *liczby_pierwsze.dll* należy wydać dwa polecenia:

- `gcc -c liczby_pierwsze.c` (kompiluje do tzw. *object code*)
- `gcc -shared -o liczby_pierwsze.dll liczby_pierwsze.o` (kompilacja do biblioteki .dll)

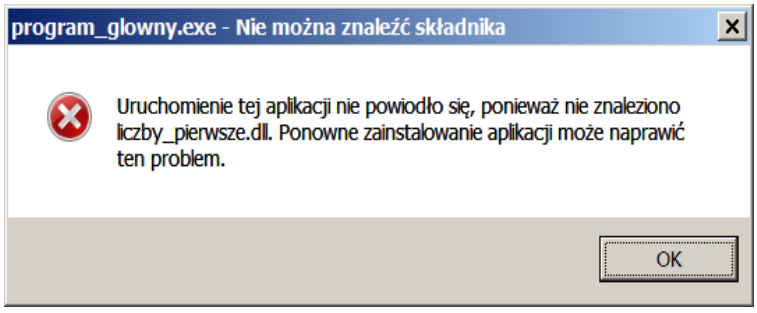
Ich dokładne znaczenie można znaleźć w dokumentacji kompilatora GCC, np. pod adresem http://gcc.gnu.org/onlinedocs/ . Warto może jednak wyjaśnić krótko czym jest *object code*. Kompilator jest w stanie przeprowadzić kompilację kodu programu, bez dołączania bibliotek (bez tzw. linkowania). W przypadku tego rodzaju kompilacji powstaje wynikowy plik binarny, tzw. *object*, który zawiera kod wykonywalny programu.

Następnie należy skompilować *program_glowny.c* wykorzystując opcję dołączenia biblioteki zewnętrznej *liczby_pierwsze.dll*:

- `gcc prorgam_glowny.c -o program_glowny.exe -I./ liczby_pierwsze.h -L./ -lliczby_pierwsze`

gdzie polecenie -I./ dołącza plik nagłówkowy *liczby_pierwsze.h* z deklaracją funkcji bibliotecznej, a polecenie -L./ jest dyrektywą dla linkera skojarzenia pliku wykonywalnego prorgamu z biblioteką *liczby_pierwsze.dll*.

W wyniku pomyślnie przeprowadzonej kompilacji powstanie plik wykonywalny *prorgam_glowny.exe*. Wykonując eksperyment, plik ten można umieścić w innym katalogu i spróbować go uruchomić. Jeżeli uruchomimy go w separacji od pliku *liczby_pierwsze.dll*, otrzymamy komunikat informujący o zależności naszego programu od tego pliku:



Omawiany program korzysta z funkcji bibliotecznej `sitoEratostelesa()` i do poprawnego działania wymaga dostarczenia biblioteki w której ona rezyduje. Jeżeli umieścimy oba pliki w jednym katalogu i ponownie uruchomimy program, zostanie wydrukowany stosowny zbiór liczb pierwszych.

Przedstawiony tu sposób tworzenia i wykorzystania bibliotek odnosi się do tzw. bibliotek dynamicznych o których będzie mowa w punkcie 2.6.

2.6 Wykorzystanie bibliotek zewnętrznych

Biblioteki zewnętrzne to takie, które zawierają kolekcje niestandardowych funkcji użytkownika i nie są standardowo włączane do kompilatora. Dla języka C istnieje wiele bibliotek zewnętrznych, dzięki którym otrzymujemy gotowe funkcje do realizacji takich typowych zadań jak:

- tworzenie graficznych interfejsów użytkownika, aplikacji sieciowych i wszelkiego oprogramowania użytkowego pod kontrolą MS Windows np.: Windows API
- tworzenie animacji trójwymiarowych, np.: OpenGL
- kompresowanie plików, np.: Zlib, ZipArchive
- tworzenie i obsługa lokalnej bazy danych, np.: sqlite
- tworzenie dokumentów PDF, np.: Haru Free PDF Library
- kryptografia, np.: OpenSSL
- metody numeryczne, obliczenia inżynierskie, np.: GNU Scientific Library
- kodery i dekodery formatów audio,np: MP3' Tech, Bass
- kodery dekodery formatów wideo, np.: FFmpeg

Biblioteki mogą funkcjonować jako statyczne lub dynamiczne. Biblioteka statyczna to taka, która jest łączona z programem w momencie linkowania (konsolidacji). Taki sposób łączenia nazywany jest *łączeniem statycznym*. Pliki binarne takich bibliotek mają typowo rozszerzenia: *.lib* (kompilator Visual C++), *.obj* oraz *.o, .a* (kompilator MinGW/GCC). Biblioteka dynamiczna jest łączona z programem wykonywalnym w momencie jego uruchomienia. W takim przypadku to system operacyjny będzie obsługiwał mechanizm łączenia. Technika ta nazywana jest *łączeniem dynamicznym*. Pliki binarne bibliotek dynamicznych w systemach MS Windows mają rozszerzenie *.dll*.

W punkcie 2.5 został już omówiony proces tworzenia biblioteki użytkownika. Przedstawione tam ćwiczenie miało na celu pokazanie przede wszystkim ogólnej zasady użycia w programie funkcji zawartych w bibliotece. Taka metoda jest jak najbardziej poprawna z punktu widzenia aplikacji, która stanowi produkt finalny udostępniany użytkownikowi i zaleca się jej stosowanie.

Podczas programowania i uruchamiania aplikacji korzystających z bibliotek zewnętrznych stosuje się niekiedy praktykę dołączania statycznego. Wówczas program i biblioteka stanowią jeden plik. W przypadku bibliotek standardowych jest to często domyślny sposób ich dołączania do pliku wykonywalnego. Sposób generacji kodu i dołączania można wymusić odpowiednim ustawieniem kompilatora i linkera. Działania takie podejmuje się raczej w przypadku, gdy tworzony program jest kompilowany w jednym środowisku a uruchamiany w innym. Zauważmy jednak, że takie działanie powoduje zwiększenie rozmiaru pliku wykonywalnego programu. Jakkolwiek z oczywistych powodów łączenie statyczne może być właściwie dla bibliotek standardowych C, w przypadku bibliotek zewnętrznych nie powinno się go stosować.

Wykonajmy ćwiczenie pokazujące proces uruchamiania programu korzystającego z funkcji znajdującej się w bibliotece zewnętrznej dołączanej statycznie. W przykładzie wykorzystamy program sito Eratostenesa, jak w punkcie 2.5. Plik binarny biblioteki statycznej tworzony prze kompilator GCC ma rozszerzenie .o. Przypomnijmy, że plik taki był potrzebny pośrednio do utworzenia biblioteki .dll. Zatem w celu utworzenia biblioteki statycznej wystarczy skompilować plik *liczby_pierwsze.c* z opcją *-c*:

- `gcc -c  liczby_pierwsze.c -o liczby_pierwsze.o.`

Po wykonaniu polecenia otrzymujemy plik *liczby_pierwsze.o,* któremu można nadać bardziej zwięzłą nazwę: *liblpierwsze.o*. Linker skojarzony z kompilatorem GCC rozpoznaje pliki binarne bibliotek po pierwszych trzech literach nazwy – co tłumaczy wyżej przyjętą konwencję. Typowa biblioteka statyczna obsługiwana przez GCC powinna mieć formę skompresowanego archiwum. Wygenerujemy je korzystając z programu archiwizującego skojarzonego z kompilatorem GCC:

- `ar rcs libpierwsze.a libpierwsze.o`

Informacje o opcjach powyższego wywołania są dostępne po wywołaniu programu **ar** bez parametrów.

Spróbujmy skompilować teraz *program_glowny.c* dodając opcję łączenia utworzonej biblioteki statycznej:

- `gcc program_glowny.c -o program_glowny.exe -L./ -llpierwsze`

Tym sposobem otrzymujemy plik wykonywalny .exe. Co odróżnia go od programu uzyskanego w przykładzie z punktu 2.5? Plik .exe może być uruchamiany niezależnie od powiązanych z nim bibliotek - z dowolnego miejsca - gdyż biblioteka *liblpierwsze.a* została do niego dołączona.

2.7 Uruchamianie złożonych programów

Pracę nad tworzeniem kodu i uruchamianiem złożonych aplikacji zwykle wykonuje się w zintegrowanym środowisku programistycznym IDE. Poza umiejętnością tworzenia kodu istotna staje się umiejętność wykorzystania narzędzi oferowanych przez środowisko. Dzięki IDE uzyskamy m.in.:

- szybszy dostęp do typowych opcji podstawowych narzędzi typu: kompilator, linker itd.
- łatwe uruchomienie kompilatora (zwykle kombinacja klawiszy F9, ctrl + F9 lub F6)
- szybszą korekcję błędów
- łatwiejsze debugowanie; debugowanie polega ogólnie na krokowym wykonywaniu programu i śledzeniu jego przebiegu w celu wykrycia błędów logicznych w jego działaniu
- sprawne testowanie

2.7.1 Praca z kompilatorem i środowiskiem IDE

Istnieje wiele różnych środowisk programistycznych wspierających język C. Najbardziej popularne komercyjne IDE dla systemów MS Windows to:

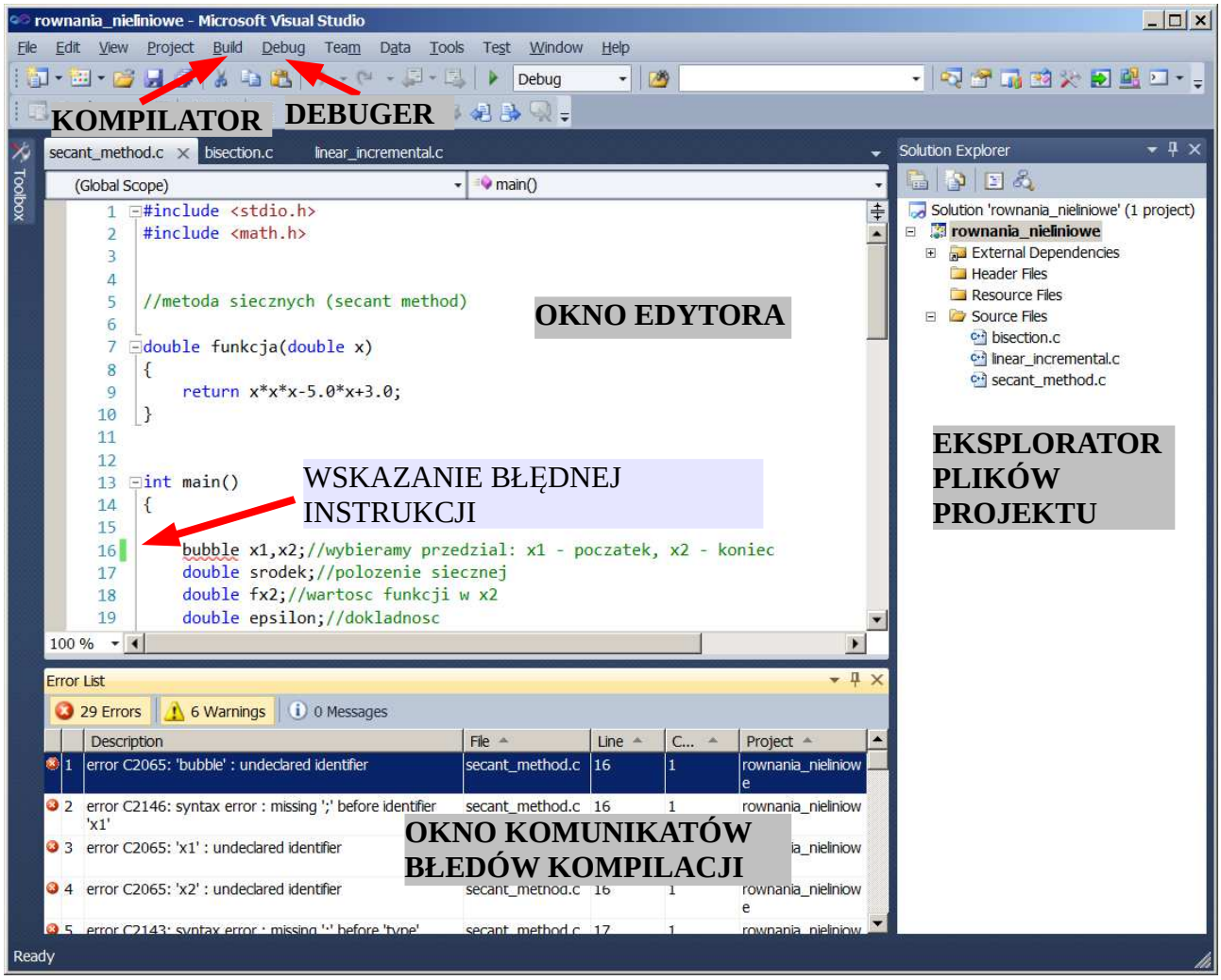
- Microsoft Visual Studio
- Borland C++ Builder (współcześnie już nie rozwijany, zastąpiony)

Wśród popularnych środowisk niekomercyjnych najczęściej wykorzystywane są:

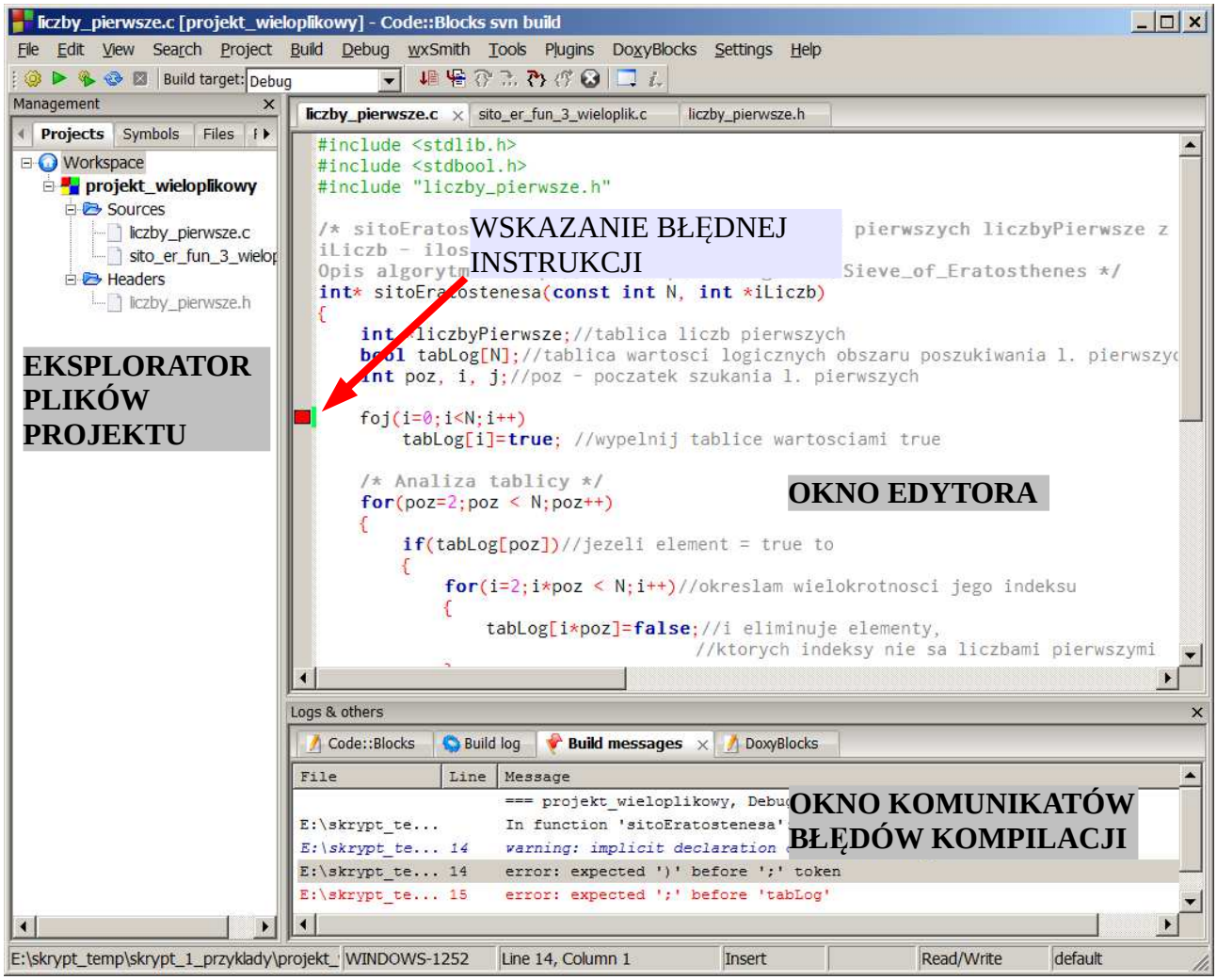
- Eclipse – bardzo popularne środowisko napisane w Javie, na bazie którego tworzy się również specjalizowane IDE w wydaniu komercyjnym; w przypadku języka C odmiana Eclipse CDT
- Microsoft Visual Studio Express
- NetBeans ze wsparciem dla C
- Code::Blocks
- Dev-C++

Poniżej przedstawiono przykładowe ekrany główne dwóch wybranych IDE: Microsoft Visual Studio oraz Code::Blocks wraz ze wskazaniem elementów wspólnych. Środowisko Visual Studio posiada wbudowany kompilator C/C++ wspierający autorskie rozszerzenia firmy Microsoft. Niestety nie jest w pełni kompatybilny z normą C99. Środowisko to pozwala na szybkie wizualne tworzenie graficznych interfejsów użytkownika, nowoczesnych aplikacji zarówno lokalnych jak i sieciowych w technologiach WindowsForms, ASP.NET (biblioteka do tworzenia aplikacji internetowych) i ADO.NET (biblioteka do obsługi baz danych). W tym celu zostało wyposażone w potężną bibliotekę obiektów .NET. Inne języki wspierane przez Visual Studio to C#, F#, J# oraz VisualBasic. Środowisko to wspiera wiele nowatorskich technologii, których nie starano się nawet wymienić mając na uwadze ramy niniejszego opracowania

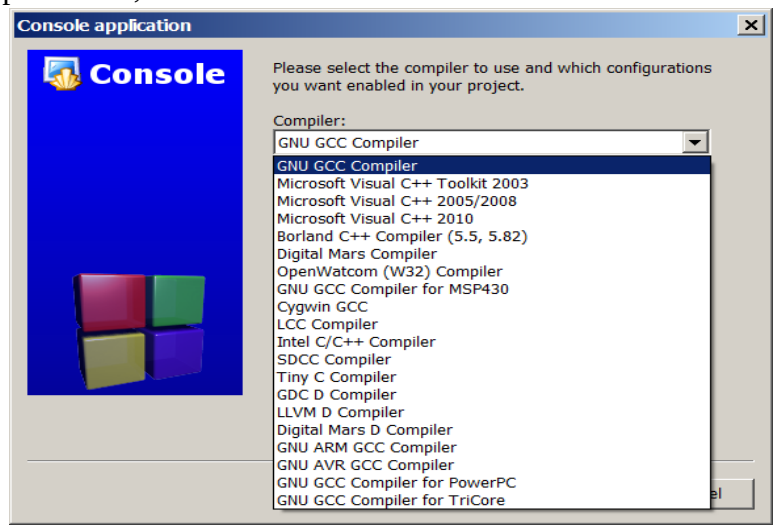
Ekran główny Microsoft Visual Studio



Ekran główny Code::Blocks



Środowiska niekomercyjne typowo integrowane są z różnymi odmianami kompilatora GCC (the GNU Compiler Collection). Cechą wyróżniającą Code::Blocks jest to, że potrafi współpracować również z innymi kompilatorami, oto ich lista:

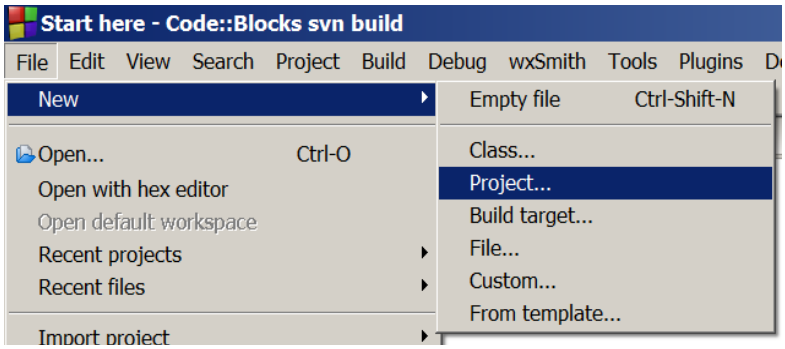


W odróżnieniu od Microsoft Visual Studio, Code::Blocks jest przeznaczone typowo do pracy z kompilatorami języka C/C++. Jest sukcesywnie rozwijane na licencji wolnego oprogramowania w ramach projektu programistycznego www.codeblocks.org.

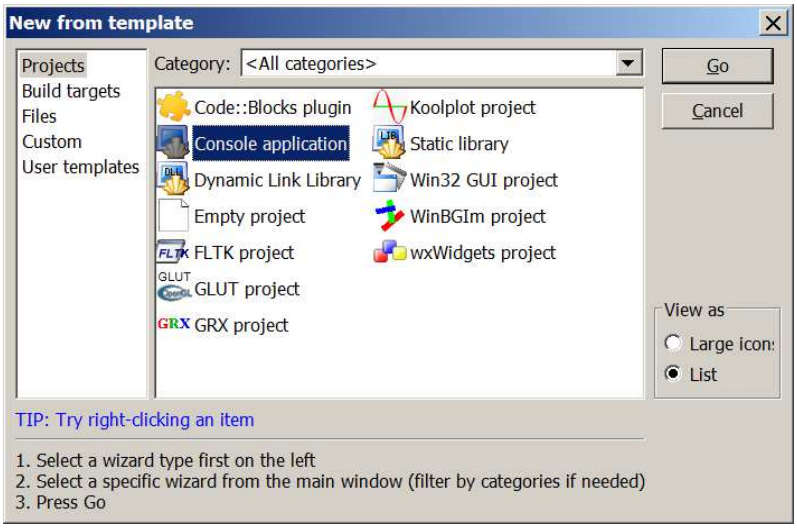
Zaprezentowane środowiska IDE promują koncepcję pracy nad tworzonym oprogramowaniem w ramach wydzielonej przestrzeni roboczej, zawierającej się w ustalonym katalogu. W Code::Blocks mamy możliwość pracy nad wieloma programami jednocześnie, organizując je w postaci tzw. projektów skojarzonych z przestrzenią roboczą, tzw. workspace. W Microsoft Visual Studio rolę przestrzeni roboczej pełni tzw. solucja (ang. solution) lub poprawniej – rozwiązanie zadania. Wśród wielu rodzimych użytkowników tego środowiska używany jest spolszczony termin *solucja* lub po prostu *solution*. W ramach *solution* można tworzyć lub dodawać projekty. Niżej zaprezentowano czynności związane z tworzeniem i uruchomieniem w obu środowiskach IDE prostego programu jednoplikowego.

Tworzenie projektu w Code::Blocks

Pracę z Code::Blkocks rozpoczynamy od utworzenia projektu (*File → New → Project*):

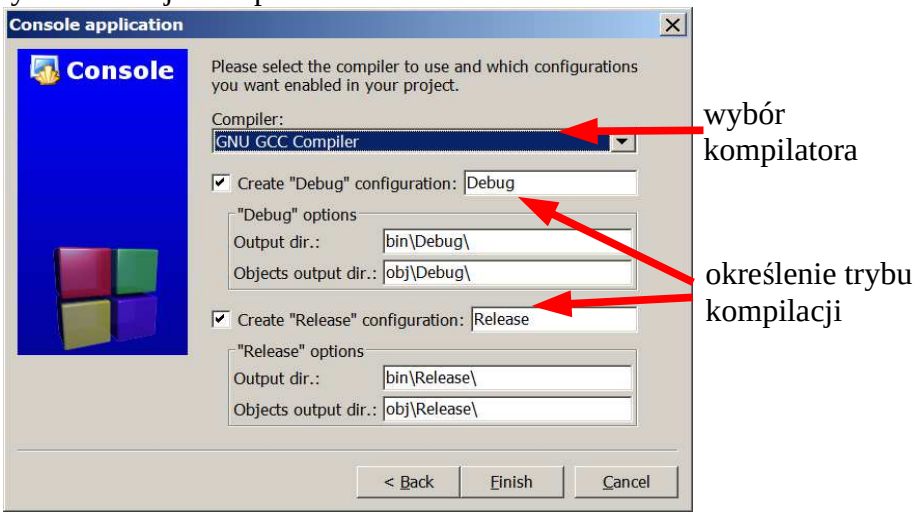


Zależnie od wersji Code::Blocks, mamy możliwość wyboru typu projektu (*Console application*):

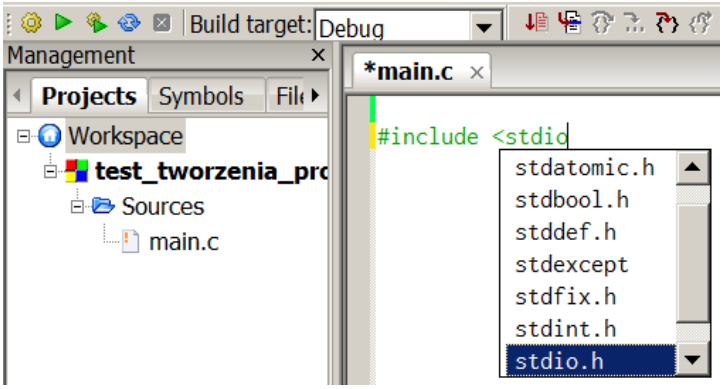


Prostym programem C można typowo zarządzać w ramach projektu "Console aplication". Proces tworzenia projektu w tym środowisku ma na celu przede wszystkim ustalenie katalogu w którym będą rezydowały pliki z kodami źródłowymi oraz stosowne ustawienie kompilatora (np. ważne przy bardziej skomplikowanych zadaniach dołączenie wymaganych bibliotek niestandardowych). W trakcie tworzenia projektu podaje się miejsce utworzenia katalogu projektu oraz jego nazwę, która

zwykle pokrywa się z nazwą katalogu projektu. Wybiera się także tryb kompilacji (Debug – z generacją informacji dla debugera, oraz Release – wersja finalna programu). Wedle potrzeby, istnieje możliwość wyboru rodzaju kompilatora.



Pliki źródłowe do prostego projektu aplikacji konsolowej dodaje się ręcznie, wybierając: File → New → Empty file. Zostaniemy przy tym poproszeni o podanie nazwy pliku. W przypadku tworzenia innego rodzaju projektu, środowisko typowo automatycznie generuje pliki źródłowe, tworzące przykładowy szablon szablon aplikacji.



Po wprowadzeniu kodu źródłowego programu, skompilujemy go wybierając opcję *Build* z menu *Build*. Kompilator można również uruchomić kombinacją klawiszy **Ctrl** i **F9**. Skompilowany program uruchamia się z poziomu IDE kombinacją klawiszy **Ctrl** i **F10**, lub opcją *Run* z menu *Build*.

Środowisko posiada kilka cech charakterystycznych dla rozbudowanych komercyjnych IDE:

- automatycznie organizuje pliki źródłowe projektu, włączając je do kategorii *Sources*, a pliki nagłówkowe umieszczając w kategorii *Headers*
- posiada różne schematy kolorowania składni
- zawiera narzędzie sugerowania słów kluczowych, nazw standardowych funkcji bibliotecznych lub skojarzonych funkcji zewnętrznych
- automatycznie tworzy katalog dla nowo zakładanego projektu

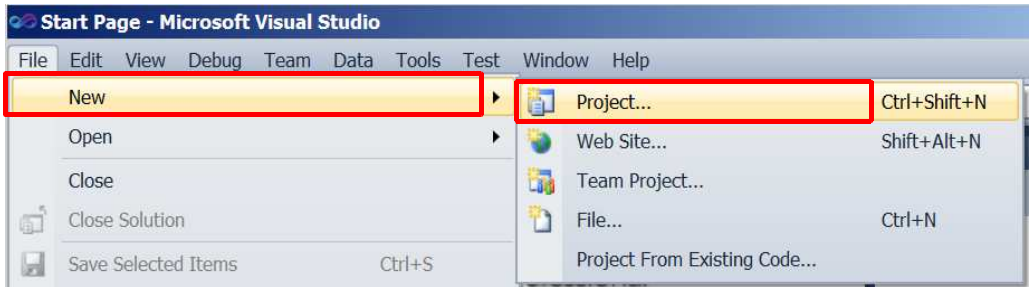
Code::Blocks jest środowiskiem bardzo uniwersalnym a przy tym prostszym w obsłudze niż np. Eclipse czy Microsoft Visual Studio. W razie potrzeby można je zainstalować w wersji bez

kompilatora. Wówczas mamy możliwość skojarzenia IDE z kompilatorem zainstalowanym poza środowiskiem. Przynosi to podstawową korzyść: aktualizację kompilatora oraz IDE można wykonać niezależnie.

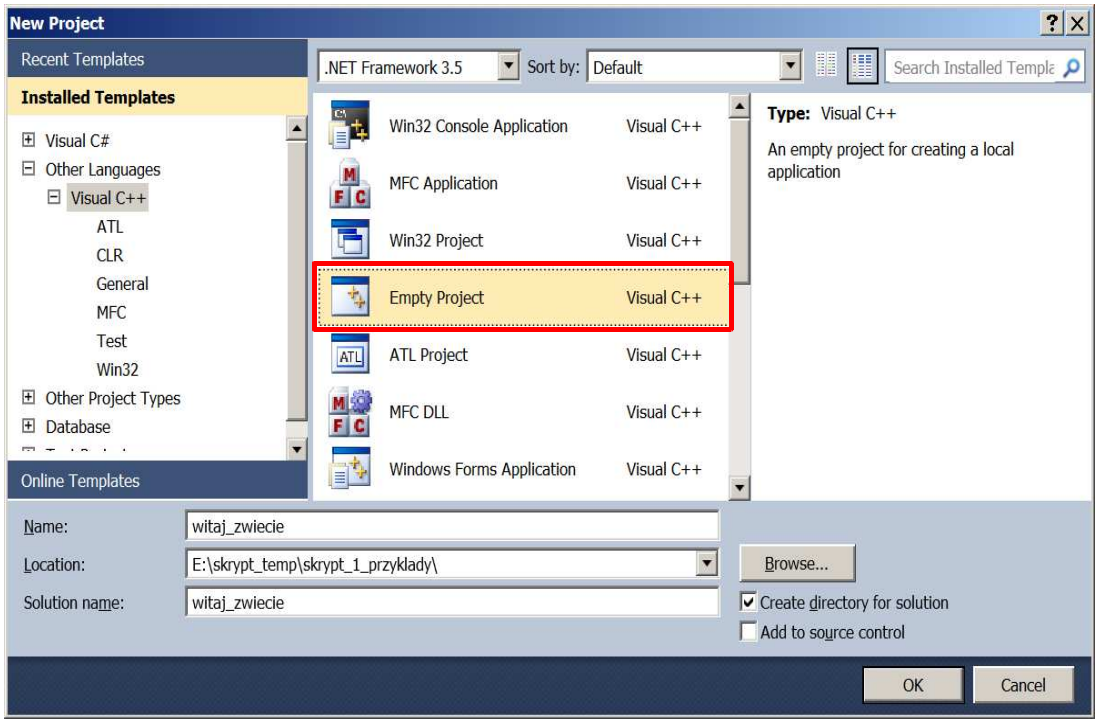
Choć nie jest to polecaną praktyką, Code::Blocks pozwala na kompilację kodów źródłowych znajdujących się w pliku nie skojarzonym z projektem. Wystarczy po prostu otworzyć taki plik w programie i uruchomić kompilator. W takim przypadku nie mamy jednak możliwości debugowania programu z poziomu IDE. Poza tym, gdyby uruchamiany program wymagał dołączenia niestandardowych bibliotek, opcję tą należy ustawić globalnie w ustawieniach kompilatora. Może to powodować niekiedy kompilkacje, gdy naprzemiennie pracujemy nad wieloma programami korzystającymi z różnych niestandardowych bibliotek.

Tworzenie projektu w Microsoft Visual Studio

Podobnie jak w przypadku Code::Blocks pracę zaczynamy od utworzenia projektu:

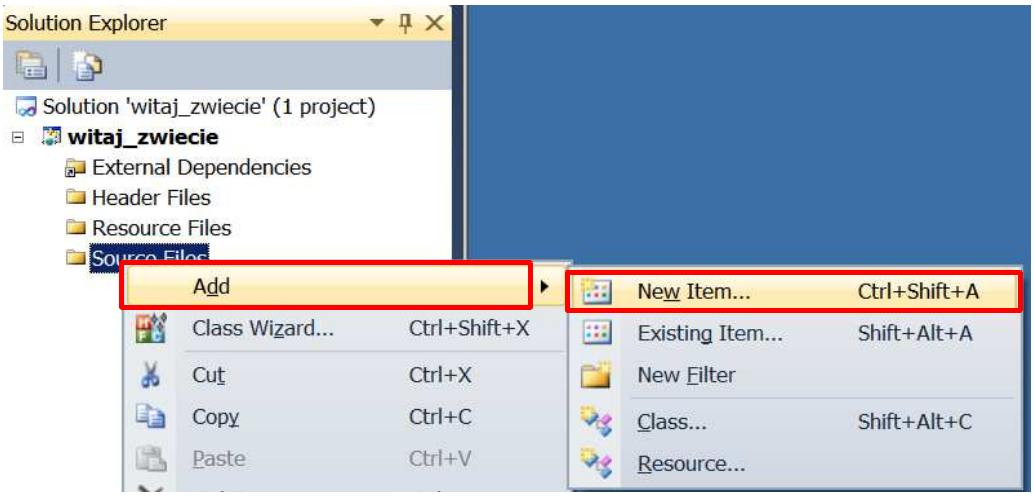


W Visual Studio wbudowano wiele szablonów standardowych projektów. Domyślnie środowisko to nie wspiera możliwości tworzenia projektów aplikacji kodowanych w języku C. Można jednak w tym przypadku wybrać pusty szablon projektu Visual C++ (Empty project):

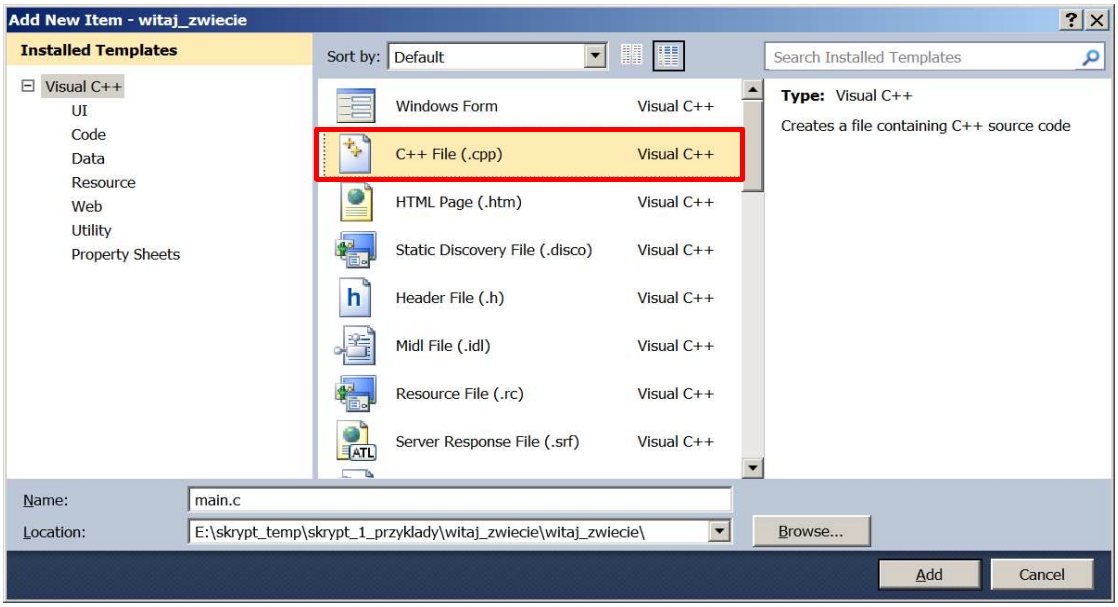


Po nadaniu nazwy dla projektu, tworzony jest domyślnie katalog przeznaczony na pliki źródłowe oraz inne powiązane.

Po utworzeniu projektu, pliki z kodem źródłowym oraz nagłówki dodajemy ręcznie. W przypadku konieczności dodania plików źródłowych języka C , rozwijamy menu podręczne (klik prawym klawiszem myszy) dotyczące plików źródłowych Source files:



Następnie wybieramy rodzaj pliku źródłowego "C++ File (.cpp)", nadajemy mu nazwę oraz rozszerzenie .c (nie jak sugeruje IDE .cpp) . Poprzez ustalenie rozszerzenie .c dla pliku źródłowego ustawiamy domyślnie kompilator w tryb kompilacji kodu C.



Visual Studio posiada mechanizm IntelliSense - bieżącej analizy poprawności niektórych składników kodu, przede wszystkim zgodności typów oraz poprawności wywołania funkcji bibliotecznych i lokalnych funkcji użytkownika. W wielu przypadkach mechanizm ten pozwala na poprawienie typowych błędów sematycznych, bez konieczności kompilacji kodu.

Kompilator w Visual Studio uruchamiamy z wybierając *Build solution* z menu *Build*. W zależności od ustawień domyślnych, proces kompilacji można zainicjować również klawiszem *F6*. Uruchomienie skompilowanego prorgamu wykonamy opcją *Start Without Debugging* (Ctrl i F5) z

menu *Debug*.

Visual Studio jest bardzo rozbudowanym IDE, przez co może sprawiać wrażenie trudnego w obsłudze, szczególnie dla niedoświadczonych użytkowników. W opinii Autora obsługa IDE w przypadku konieczności uruchamiania prostych przykładów akademickich, czy nawet projektów wieloplikowych jest równie mało skomplikowana jak w Code::Blocks lub w popularnym wśród studentów Dev-C++. Gwarancją powodzenia przy korzystaniu z każdego IDE jest zapoznanie się z jego podstawowymi możliwościami i postępowanie według wyżej opisanych reguł.

2.7.2 Detekcja błędów i testowanie

Wykrywanie i poprawianie błędów jest czynnością żmudną a często również zniechęcającą początkujących programistów. Złożoność programu jest uzależniona od liczby powiązań między jego składowymi a w oprogramowaniu jest zazwyczaj wiele składowych i interakcji między nimi. Wykrywanie błędów jest tak istotnym zadaniem, że firmy produkujące oprogramowanie posiadają całe zespoły osób odpowiedzialnych za testowanie i poprawianie kodu.

Wykrywanie błędów jest trudne i może trwać nieprzewidywalnie długo. Doświadczeni programiści wiedzą, że tworzenie kodu zwykle trwa tyle samo co poprawianie w nim błędów. Można nieco skrócić proces korekcji błędów, jeżeli: programuje się w dobrym stylu, projekt został zweryfikowany przed rozpoczęciem kodowania, korzysta się z narzędzi wspomagających sprawdzanie kodu, używa się niewielu zmiennych globalnych itd.

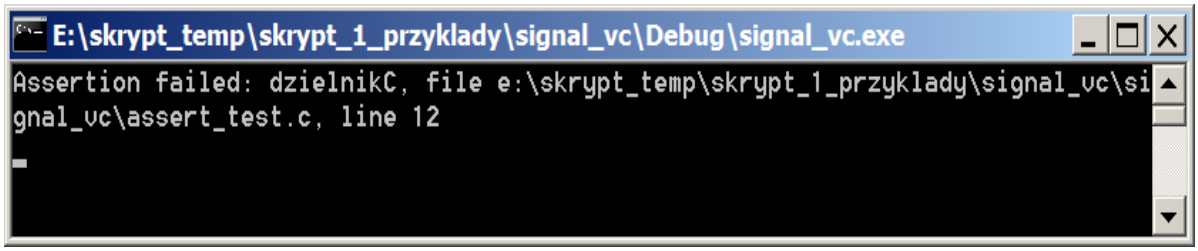
Niektóre języki wyższego poziomu jak Java czy C# posiadają mechanizmy zapobiegające typowym błędom, jak: sprawdzanie zakresów indeksów, brak wskaźników, automatyczne odśmiecanie pamięci (tzw. garbage collector) oraz rygorystyczne sprawdzanie zgodności typów. Właściwości języka zapobiegające typowym błędom mają jednak swoją cenę. Jeżeli język pozwala na eliminację prostych błędów, to zazwyczaj narażony jest na błędy wysokiego poziomu.

Wyróżnia się dwa zasadnicze rodzaje błędów:

- błędy składniowe (ang. syntax error), związane z niepoprawną składnią, nie zachowaniem reguł danego języka
- błędy logiczne (ang. logic error), związane z niepoprawnym działaniem programu

Źródła programu, który zawiera błędy składniowe prawdopodobnie nie da się skompilować, choć w skrajnych przypadkach jest to możliwe. Na błędy tego rodzaju zwraca uwagę kompilator lub ewentualnie linker, więc można je z powodzeniem poprawić. Kod zawierający błędy logiczne a nie zawierający składniowych, daję się skompilować. Z tego powodu błędy logiczne nawet w prostych programach mogą być bardzo trudne do weryfikacji. Wówczas jedyną drogą ratunku może być wykorzystanie programu uruchomieniowego potocznie nazywanego debugerem. Wykorzystanie tego typu oprogramowania zostanie omówione w kolejnym punkcie.

Język C niestety nie posiada zaawansowanych mechanizmów reakcji na błędy logiczne. Jest jednak możliwość sterowania sposobem wykonywania programu w przypadku zaistnienia sytuacji wyjątkowej, dzięki makru `assert()` czy `exit()`. Asercja jest deklaracją czegoś, co powinno być prawdziwe. W czasie wykonywania makro `assert()` testuje, czy jego argument jest niezerowy. Jeżeli jest zerowy, wysyła wiadomość diagnostyczną typu:



Spróbujmy napisać prostą funkcję zwracająca wynik dzielenia dwóch liczb rzeczywistych, pamiętając o wykluczeniu dzielenia przez zero. Zadanie to łatwo można wykonać wstawiając warunek zapobiegający dzieleniu, gdy dzielnik = 0. Można jednak wykorzystać assert() w roli nadzorcy poprawności dzielenia:

```
1.  #include <stdio.h> /* funkcje: fprintf(), printf() */
2.  #include <stdlib.h> /* funkcja: exit() */
3. #include <assert.h>
4.
5.  int main( void )
6. {
7.     int dzielnaC = 50;//dzielna calkowita
8.     int dzielnikC = 0;//dzielnik calkowity
9.     int ilorazC;//iloraz calkowity
10.
11.        assert(dzielnikC);
12.
13.        printf("%d / %d = %d\n", dzielnaC, dzielnikC, ilorazC = dzielnaC/dzielnikC);
14.
15.     exit(EXIT_SUCCESS); /* Pomyslne zakonczenie programu */
16. }
```

Powyższy program generuje komunikat błędu asercji, przedstawiony wyżej przy opisie funkcji `assert()`.

Bardziej zaawansowaną obsługę sytuacji wyjątkowych zrealizujemy przy użyciu funkcji typu `signal()` (biblioteka *signal.h*). Funkcja `signal` posiada dwa parametry: typ sygnału, oraz wskaźnik do funkcji, która ma zostać wywołana, gdy zostanie przechwycony dany sygnał. Typowe sygnały rozpoznawane przez wymienioną funkcję to:

- **`SIGABRT`** – program zażądał wyjątkowego przerwania pracy
- **`SIGFPE`** – pojawił się błąd arytmetyczny
- **`SIGILL`** – wykryto nieprawidłową instrukcję
- **`SIGINT`** – przechwytuje kombinację klawiszy Ctrl i C powodującą zakończenie programu UWAGA: według dokumentacji Microsoft, wywołanie zakończenia programu powoduje utworzenie dodatkowego wątku, wówczas aplikacja jedno wątkowa staje się aplikacją wielowątkową; może prowadzić to do nieprzewidzianego zachowania programu
- **`SIGSEGV`** – wykryto nieprawidłowy dostęp do pamięci
- **`SIGTERM`** – otrzymano żądanie przerwania działania programu

Jako przykład obsługi sygnałów przeanalizujemy działanie bardziej wyrafinowanego programu wykonującego w efekcie swojego działania naruszenie pamięci poprzez przypisanie wartości 0 do pustego wskaźnika:

```
1. #include <stdio.h>
2. #include <stdlib.h>//exit()
3. #include <signal.h>//signal() przechwytywanie i obsluga sygnalow
4.
5. void wyjscie(int sig);
6.
7. int main()
8. {
9.       int *x = NULL;
10.       signal(SIGSEGV, wyjscie);
11.       *x = 0;//;proba naruszenia pamieci
12.       exit(EXIT_SUCCESS);
13. }
14. void wyjscie(int sig)
15. {
16.       printf("Blad dostepu do niezaalokowanej pamieci! Dowolny klawisz - wyjscie...\n");
17.       getchar();
18.       exit(sig);
19. }
```

Zgodnie z przypuszczeniem, funkcja `signal()` zareaguje na sytuację naruszenia niezaalokowanej pamięci. Błąd został spowodowany przez przez instrukcję `*x = 0;` . W tym przypadku zamiast zawieszenia działania programu wywoływana jest funkcja `wyjscie()`, która powoduje wyświetlenie stosownego komunikatu i awaryjne opuszczenie programu.

Warto pamiętać, że gdy pojawi się sygnał system operacyjny najpierw przeprowadza domyślną akcję obsługi sygnału. Stąd nie zawsze będzie skuteczna procedura obsługi sygnału, zawarta w programie. Można wówczas podjąć próbę generacji sygnału z poziomu programu, używając funkcji `raise()`. Jako ostatni przykład w tym punkcie przedstawimy ilustrację użycia raise(). Funkcja `signal()` nasłuchuje nadejścia `SIGFPE` generowanego sztucznie przez `raise(SIGFPE)` w przypadku, gdy warunek `dzielnik==0` est prawdziwy. Instrukcja `printf("%d",dzielna/dzielnik);` w przedstawionym kontekście nigdy się nie wykona.

```
1. #include <stdio.h>
2. #include <stdlib.h>//exit()
3. #include <signal.h>//signal() przechwytywanie i obsluga sygnalow
4.
5. void wyjscie(int sig);
6.
7. int main()
8. {
9.       int dzielna,dzielnik;
10.       (void)signal(SIGFPE, wyjscie);
11.       dzielna=1;
```

```
12.       dzielnik=0;//spowoduje blad dzielenia przez 0
13.       if(dzielnik==0)
14.             raise(SIGFPE);//generuje sygnal bledu arytmetycznego
15.       printf("%d",dzielna/dzielnik);//ta instrukcja nie zostanie wykonana
16.       exit(EXIT_SUCCESS);
17. }
18.  void wyjscie(int sig)
19. {
20.       printf("Blad arytmetyczny! Dowolny klawisz - wyjscie...\n");
21.       getchar();
22.       exit(sig);
23. }
```

2.7.3 Zastosowania debuggera

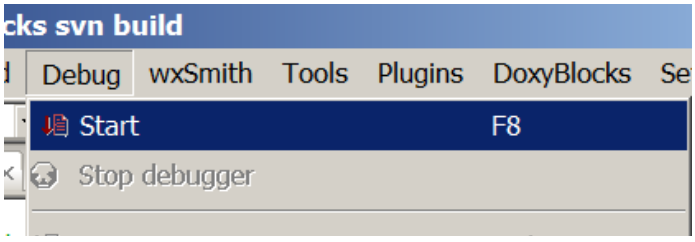
Debugowanie ma na celu wykrycie błędów w logice działania programu. Proces ten najlepiej jest przeprowadzić przy użyciu debugera graficznego, dostępnego w wielu popularnych IDE. Do zaprezentowania sposobu pracy z tym narzędziem zostało wybrane środowisko Code::Blocks.

Istota wykorzystania debugera zostanie omówiona na przykładzie prostego programu wykonującego niedozwolone dzielenie przez 0:

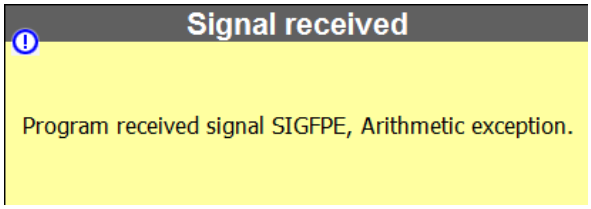
```
1.  #include <stdio.h> /* funkcje: fprintf(), printf() */
2.  #include <stdlib.h> /* funkcja: exit() */
3.
4.  int main( void )
5.  {
6.     int dzielnaC = 50;//dzielna calkowita
7.     int dzielnikC = 0;//dzielnik calkowity
8.     int ilorazC;//iloraz calkowity
9.
10.       printf("%d / %d = %d\n", dzielnaC, dzielnikC, ilorazC=dzielnaC/dzielnikC);
11.
12.     exit(EXIT_SUCCESS); /* Pomyslne zakonczenie programu */
13. }
```

Po wykonaniu kompilacji zaleca się uruchomienie programu. Następstwem uruchomienia będzie zawieszenie wykonywania programu, spowodowane wygenerowaniem błędu dzielenia przez 0. Spróbujmy przyjąć, że nie mamy pojęcia o przyczynie nieprawidłowego działania programu i chcemy wykryć usterkę przy użyciu wbudowanego debugera.

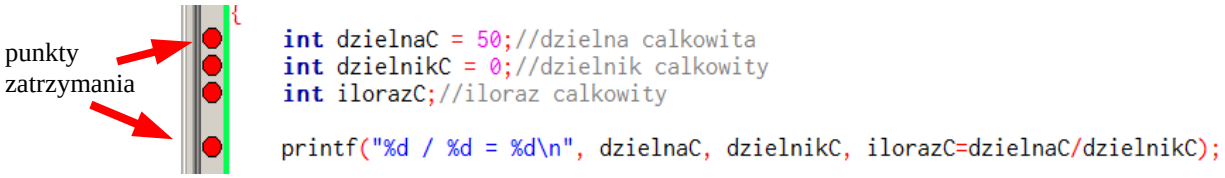
W tym celu wybieramy opcję uruchomienia debugera (*Debug* → *Start* lub *F8*):



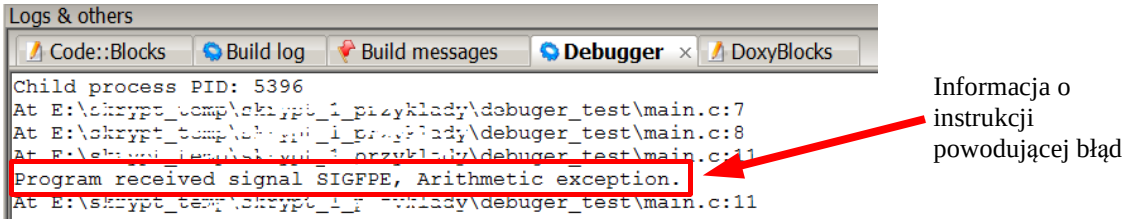
Uruchomienie programu w tym trybie spowoduje wyświetlenie komunikatu, w którym zawarta jest informacja o wykryciu sygnału błędu arytmetycznego SIGFPE:



Znamy więc już przyczynę niepoprawnego działania naszego programu, wyłączmy więc debuger wybierając opcję *Stop debugger* z menu *Debug*. Spróbujmy teraz określić miejsce występowania instrukcji, która powoduje błąd. Debuger posiada opcję uruchomienia programu w trybie krokowym. W tym trybie można wymusić wykonywanie programu instrukcja po instrukcji. Aby skorzystać z tej opcji, należy zaznaczyć instrukcje, które mają być wykonane w danym kroku (czyli instrukcje na których ma zatrzymać się wykonywanie programu w danym kroku). W tym celu dodamy do programu tzw. punkty zatrzymania (ang. break point). W Code::Blocks wystarczy kliknąć obok instrukcji w lewą krawędź okna edytora. Pojawiające się czerwone punkty świadczą o wstawieniu w danym miejscu punktu zatrzymania:



Jeżeli teraz ponownie uruchomimy debuger, wykonywanie programu zostanie zatrzymane na pierwszej instrukcji zawierającej punkt zatrzymania, czyli `int dzielnaC = 50;` . Naciskając klawisz F7 spowodujemy wykonanie tej instrukcji i zatrzymanie się na kolejnej `int dzielnikC = 0;` itd. W ten sposób można wykonywać program, obserwując przy wykonaniu której instrukcji pojawi się błąd. Warto również spoglądać na okno informacji debugera, znajdujące się poniżej okna edytora:

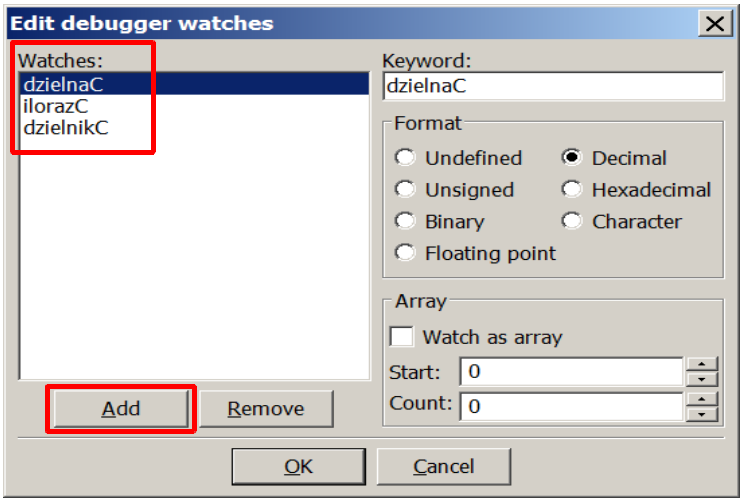


W naszym programie błędna jest instrukcja zawarta w wierszu 11. Możemy zatem wyłączyć debuger i poprawić program, eliminując dzielenie przez 0. Zauważmy, jednak że chociaż skutek błędu ma miejsce w wykonaniu dzielenia przez 0, to jednak korekty wymaga instrukcja przypisania `int dzielnikC = 0;`. Poprawimy ją inicjując zmienną `dzielnikC` wartością 2.

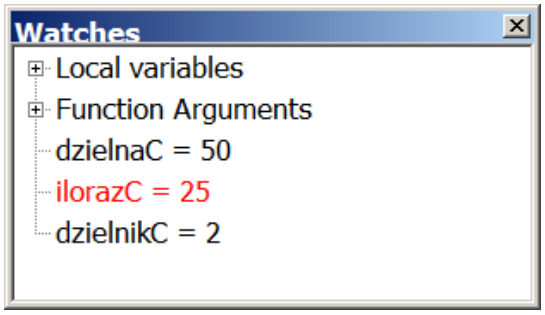
Sytuacja ta kontrastowo obrazuje typowy proces debugowania programu. Niestety nie zawsze jesteśmy w stanie celnie trafić w przyczynę błędu, lecz gdy znamy strukturę programu możemy drogą dedukcji odnaleźć właściwą ścieżkę poszukiwań.

Podczas procesu debugowania czasem istotny jest podgląd na wartości zmiennych, wśród których spodziewamy się kandydata mającego wpływ na niepoprawne działanie programu. Z poziomu menu *Debug* wybierając opcję *Edit watches..* można włączyć narzędzie dodawania zmiennych, dla których chcemy obserwować bieżące wartości w trakcie wykonywania programu.

W tym celu wystarczy wybrać polecenie *Add* i wprowadzić nazwy zmiennych:



Spróbujmy ponownie uruchomić debuger. W trakcie wykonywania kolejnych kroków programu możemy teraz obserwować bieżące wartości zmiennych:



3. Praktyczne przykłady programowania strukturalnego

W niniejszym rozdziale zostaną zaprezentowane przykłady pokazujące praktyczne aspekty programowania i uruchamiania aplikacji w języku C. Poszczególne programy można zrealizować w Microsoft Visual Studio, w Code::Blocks lub w innym preferowanym środowisku prorgamistycznym.

3.2 Realizacja aplikacji serwer czasu

Zaprezentujemy projekt realizacji aplikacji prostego serwera czasu. Program ten w przypadku połączenia klienta będzie zwracał czas systemowy maszyny na której jest uruchomiony. W tym przypadku tworzymy projekt aplikacji konsolowej w Visual Studio.

Do realizacji aplikacji sieciowych w systemach MS Windows można użyć biblioteki Winsock. Do obsługi połączenia sieciowego używa się mechanizmu tzw. gniazd sieciowych. Programowa obsługa gniazd sieciowych wymaga przeprowadzenia następujących kroków:

- Inicjacja biblioteki Winsock
  - Deklaracja zmiennej `WSADATA wsaData;`

    - Wywołanie funkcji `WSAStartup(wersja, &wsaData);`
- Deklaracja gniazda typu serwer lub klient
    - socket serwer; lub socket client;
- Deklaracja struktury przechowującej informacje o adresie internetowym (host/klient)
- Powiązanie gniazda z informacją o połączeniu
    - Funkcja `bind();`
- Nasłuch – oczekiwanie na połączenie klienta
    - Funkcja `listen();`
- Akceptacja połączenia
    - Funkcja `accept();`
- Przesył lub odbiór danych
    - Funkcje `send()` lub `recv();`
- Zakończenie połączenia
    - Zamknięcie gniazda serwer funkcją `closesocket()`;
    - Zakończenie pracy z biblioteką winsock funkcją `WSACleanup();`

Inicjację oraz test biblioteki Winsock można zrealizować za pomocą programu:

```
1. #include <stdio.h>
2. #include <winsock2.h>
3.
4. /*dolaczenie biblioteki winsock*/
5. #pragma comment(lib, "Ws2_32.lib")
6.
7.  int main()
8. {
9.       WSADATA wsaData;
10.     //uruchomienie obslugi biblioteki winsock
11.     WSAStartup(MAKEWORD(2,2),&wsaData);
12.     WSACleanup();
13.       system("PAUSE");
14.       return 0;
15. }
```

Komentarz do programu:

- program wymaga dołączenia i inicjacji zewnętrznej biblioteki **ws2_32.dll**
- WSADATA – struktura zawierająca informacje o implementacji gniazd w systemie Windows; informacje te są potrzebne do poprawnej inicjacji biblioteki **ws2_32.dll**
- Wersję implementacji gniazd podaje się jako wartość typu WORD (słowo dwubajtowe)
- Funkcja WSAStartup inicjuje użycie biblioteki winsock
- Funkcja ta przekazuje do struktury WSADATA szczegóły implementacji winsock
- Funkcja zwraca ‘0’, gdy inicjacja zakończyła się powodzeniem

Niżej przedstawimy krótkie opisy funkcji stanowiących części składowe aplikacji serwera czasu. Wymienione funkcje zawierają zwykle po kilka parametrów. Odwołano się do nich w następujący sposób: `_1` – pierwszy parametr, `_2` – drugi parametr itd.

Funkcja `socket(_1, _2, _3)`:

- _1 – address familly (rodzina adresów IP):
  - typowa wartość to AF_INET; jest używane w celu sprecyzowania wersji protokołu IP (w tym przypadku IPv4)
- _2 – typ gniazda (strumieniowe/datagramowe itp.)
  - SOCK_STREAM – precyzuje typ gniazda; w tym przypadku jest to gniazdo strumieniowe, do obsługi transmisji TCP
- _3 –protokół transmisji
  - Jeżeli wartość '0' to protokół nieokreślony (usługodawca określa protokół)

Funkcja `int bind(_1,_2,_3);`

- _1 - SOCKET s; zmienna s to deskryptor gniazda, za jej pomocą obsługujemy połączenie sieciowe
- _2 - const struct sockaddr *name – wskaźnik do struktury sockaddr, przechowującej informację o adresie, nr portu itp.
- _3 - int namelen – rozmiar w bajtach struktury 'name'

Funkcja `int listen( _1,_2);`

- _1 – SOCKET s – deskryptor gniazda (serwera)
- _2 - int backlog – liczba przychodzących połączeń; SOMAXCONN – maksymalna domyślnie przyjęta liczba (brak bliższych informacji)

Funkcja SOCKET accept(_1,_2,_3);

- _1 - SOCKET s – deskryptor gniazda (serwera)
- _2 - struct sockaddr *addr – bufor przechowujący adres maszyny (klienta), która uzyskała połączenie z aplikacją serwer
- _3 - int *addrlen – rozmiar struktury addr

UWAGA: gdy nie jest istotny adres klienta, który uzyskał połączenie z nasza aplikacja, parametry _2 i _3 mogą mieć wartość NULL

Funkcja `int send(_1, _2, _3, _4 );`

- _1 - SOCKET s – deskryptor gniazda (serwer lub klient)
- _2 - const char *buf – wskaźnik na bufor z danymi do wysłania
- _3 - int len – rozmiar w bajtach danych do wysłania
- _4 - int flags – flagi uściślające zachowanie funkcji; domyślnie '0'

Funkcja `int connect(_1, _2, _3)` jest wykorzystywana w aplikacji typu klient

- _1 - SOCKET s – deskryptor gniazda połączenie w aplikacji typu klient
- _2 - const struct sockaddr *name – struktura zawierająca informację o adresie hosta oraz porcie na którym będzie nawiązywane połączenie
- _3 - int namelen – rozmiar struktury 'name'

Struktura **sockaddr_in**:

```
struct sockaddr_in{
short sin_family;
unsigned short sin_port;
struct in_addr sin_addr;
char sin_zero[8];
};
```

Opis:
sin_family - rodzina adresów (jeżeli wartość AF_INET to typowe adresy IPv4)
sin_port – port komunikacji po IP (wartość liczbowa, najlepiej powyżej 1024)
sin_zero – wypełnienie zerami, dla zachowania rozmiaru struktury sockaddr_in zgodnego z rozmiarem struktury *sockaddr*

Struktura **in_addr**:

```
struct in_addr {
union {
struct{
unsigned char s_b1,
s_b2,
s_b3,
s_b4;
} S_un_b;
struct {
unsigned short s_w1,
s_w2;
} S_un_w;
unsigned long S_addr;
} S_un;
};
```

Opis:

Struktura in_addr reprezentuje adres IP w formacie IPv4

- Adres IPv4 składa się z czterech oktetów (bajtów) oddzielonych kropkami, np.: 192.168.002.001
- Pola struktury in_addr:
    - S_un_b – adres internetowy sformatowany jako cztery porcje unsigned char
    - S_un_w – adres internetowy sformatowany jako dwie porcje unsigned short (2 słowa; 1 słowo to 2 bajty)
    - S_addr – adres sformatowany jako jedna porcja danych typu unsigned long

Funkcje do konwersji reprezentacji:

- `htons()` host to network short
- `htonl()` host to network long
- `ntohs()` network to host short
- `ntohl()` network to host long

Dlaczego konieczna jest konwersja?

- Maszyny cyfrowe (komputery) zapisują liczby w formatach:
  - Big-Endian: B34Fh (tzw. Network Byte Order)
  - lub
  - Little-Endian: 4FB3h (tzw. Host Byte Order)
- W sieciach komputerowych paczki informacji transportowane są w formacie Network Byte Order

Aplikacja serwer czasu zawiera wywołania przedstawionych wyżej funkcji bibliotecznych. Proponuje się czytelnikowi analizę kodu oraz uruchomienie aplikacji. Kompletny kod serwera czasu został zamieszczony niżej:

```
1.  #include <stdio.h>
2.  #include <winsock2.h>
3.  #include <time.h>
4.
5.  /*dolaczenie biblioteki winsock*/
6.  #pragma comment(lib, "Ws2_32.lib")
7.
8.   int main()
9.  {
10.       int i;
11.     time_t surowyczas;
12.     struct tm * timeinfo;
13.       char wiadomosc[50]={0};
14.       WSADATA wsaData;
15.       SOCKET serwer;//deklaracja gniazda
16.     SOCKADDR_IN saddr;//deklaracja struktury opisujacej polaczenie
17.       //przechwycenie czasu systemowego
18.     time ( &surowyczas );
19.     timeinfo = localtime ( &surowyczas );
20.     //wiadomosc dla uzytkownika
21.       sprintf (wiadomosc, "Czas i data: %s", asctime (timeinfo) );
22.     //uruchomienie obslugi biblioteki winsock
23.     WSAStartup(MAKEWORD(2,2),&wsaData);
24.     //utworzenie gniazda
25.     serwer = socket(AF_INET,SOCK_STREAM,0);
26.     //okreslenie wlasciwosci polaczenia
27.     memset(&saddr,sizeof(saddr),0);
28.     saddr.sin_family = AF_INET;
29.     saddr.sin_port = htons(10000);
30.     saddr.sin_addr.s_addr =  htonl(INADDR_ANY);
31.
32.     if ( bind(serwer, (SOCKADDR*)&saddr, sizeof(saddr)) != SOCKET_ERROR )
33.       {
34.             if ( listen(serwer, 1) != SOCKET_ERROR )
35.             {
36.                   for (i = 0; i < 5; i ++ ) {
```

```
37.                     SOCKET klient = accept(serwer,NULL,NULL);//gniazdo klienta
38.                     send(klient, wiadomosc, strlen(wiadomosc), 0);
39.                     closesocket(klient);
40.                 }
41.             }
42.         }
43.     closesocket(serwer);
44.     WSACleanup();
45.     system("PAUSE");
46.     return 0;
47. }
```

Program po uruchomieniu oczekuje na nawiązanie połączenia. W chwili, gdy użytkownik połączy się z komputerem na którym uruchomiono program, zostanie wysłany do niego bieżący czas systemowy. Oto schemat przedstawiający sposób obsługi programu z wykorzystaniem pojedynczej maszyny:

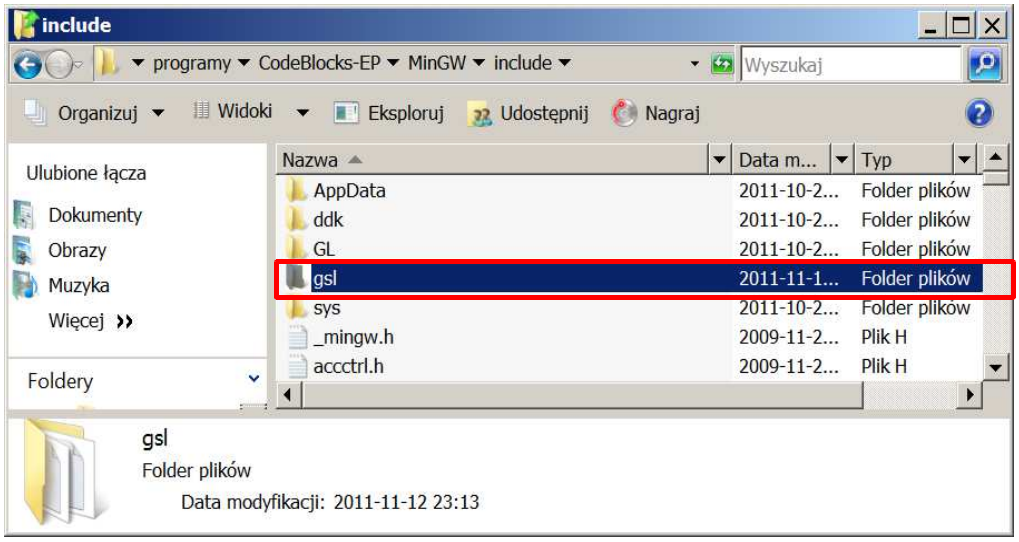
- uruchom emulator terminala telnet, np.: putty.exe
- wpisz adres IP maszyny na której jest uruchomiony program
- określ port 10000
- po uzyskaniu połączenia ujrzysz aktualny czas systemowy serwera

3.2 Przykład analizy numerycznej

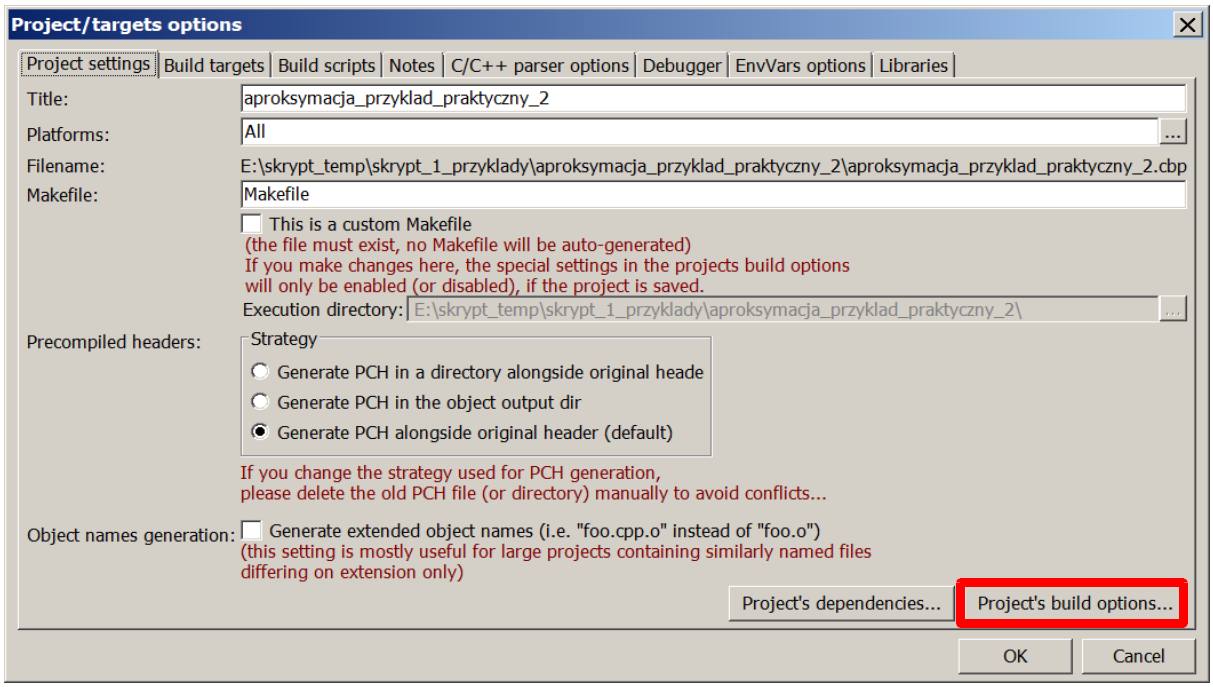
W kolejnym przykładzie zostanie pokazany program realizujący aproksymację liniową, przy wykorzystaniu zewnętrznej biblioteki GSL (GNU Scientific Library). Bibliotekę statyczną oraz skojarzone z nią pliki nagłówkowe można pobrać z witryny http://gnuwin32.sourceforge.net/packages/gsl.htm jako Developer files zawarte w paczce **gsl-1.8-lib.zip**.

W programie użyto również funkcji użytkownika, która generuje plik CSV punków zbioru danych. Program skompilowano i uruchomiono w środowisku Code::Blocks. Standardowo pracę rozpoczęto od utworzenia nowego projektu aplikacji konsolowej.

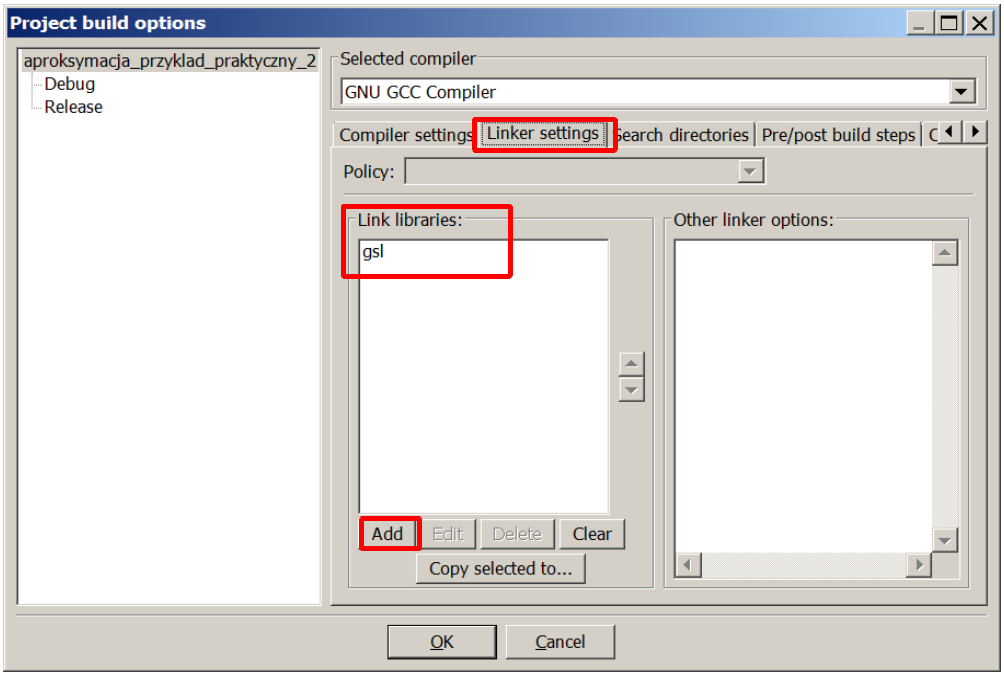
W celu umożliwienia kompilacji projektów korzystających z funkcji biblioteki GSL należy odpowiednio przygotować środowisko Code::Blocks. Zaczynamy od umieszczenia w odpowiednim miejscu pliku zawierającego bibliotekę statyczną GSL oraz skojarzone z nią pliki nagłówkowe. W katalogu głównym prorgamu Code::Blocks można odnaleźć katalog kompilatora MinGW. Pliki biblioteki znajdujące się w paczce **gsl-1.8-lib.zip** można umieścić w katalogu projektu, choć lepiej skopiować go do katalogu *lib* kompilatora MinGW. Właściwe miejsce dla plików nagłówkowych biblioteki GSL to katalog *include* kompilatora MinGW. Umieszczamy w nim cały katalog *gsl* z paczki **gsl-1.8-lib.zip**, jako pokazuje ilustracja:



Następnie należy ustawić opcję dołączenia biblioteki libgsl.a. W tym celu w Code::Blocks wchodzimy do meny *Project* i wybieramy *Properties*.... Powinno ukazać się poniższe okno, w którym wybieramy *Project's build options*...:



Następnie w zakładce *Linker settings* w polu *Link libraries* dodajemy bibliotekę *gsl* naciskając przycisk *Add*:



Pozostaje tylko skompilować i uruchomić kod źródłowy programu:

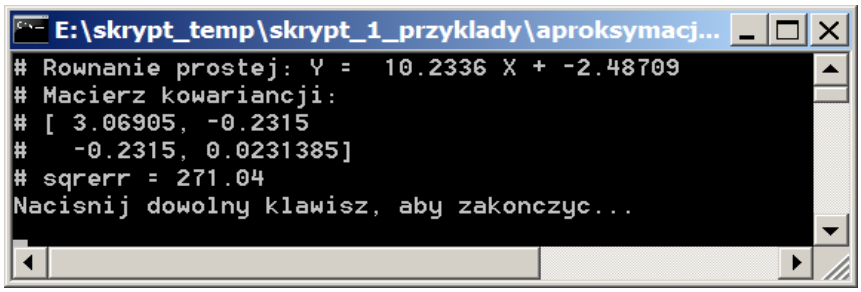
```
1. #include <stdio.h>
2. #include <math.h>
3. #include <time.h>
4. #include <stdlib.h>
5. #include <string.h>
6. #include <gsl/gsl_fit.h>
7.
8.
9.  void createCSV(double danex[], double daney[], int N, char nazwa_pliku[])
10. {
11.       int i;
12.       char tymczasowy[50];//do przechowywania sformatowanych danych (z przecinkiem zamiast
    kropki)
13.       char *str=NULL;//do dzialania funkcji strchr; funkcja wyszukuje pojedynczy znak w lancuchu
14.       FILE *fp;//wskaznik do pliku
15.
16.       fp=fopen(nazwa_pliku,"w");//otworz plik do zapisu
17.
18.
19.       for(i=0;i<N;i++)
20.       {
21.             //przygotownie danych - zamiana kropki na przecinek
22.
```

```
23.             sprintf(tymczasowy,"%lf;%lf;", danex[i],daney[i]);
24.
25.             str=strchr(tymczasowy,'.');//znajdz pierwsza kropke
26.             tymczasowy[str-tymczasowy]=',';//zamien kropke na przecinek
27.             str=strchr(str+1,'.');//znajdz druga kropke
28.             tymczasowy[str-tymczasowy]=',';//zamien kropke na przecinek
29.
30.             //zapis do pliku
31.             fprintf(fp,"%s\n",tymczasowy);//zapisz wiersz danych do pliku
32.       }
33.
34.       fclose(fp);
35. }
36.
37.  int main()
38. {
39.       double danex[20]={0};
40.       double daney[20]={0};
41.       int i,N=20;
42.
43.       double c0, c1, cov00, cov01, cov11, sqrerr;
44.
45.       srand(time(NULL));//ustaw generator liczb pseudolosowych
46.
47.       //generuj danex i daney
48.       for(i=0;i<N;i++)
49.       {
50.             danex[i]=i+0.1*(double)(rand()%10);
51.             daney[i]=10*i+(double)(rand()%10);
52.       }
53.
54.       createCSV(danex,daney,20,"daneapprox.csv");
55.
56.
57.       //aproksymacja
58.        gsl_fit_linear (danex, 1, daney, 1, N, &c0, &c1, &cov00, &cov01, &cov11,
   &sqrerr);
59.
60.        printf ("# Rownanie prostej: Y =  %g X + %g\n", c1, c0);
61.        printf ("# Macierz kowariancji:\n");
62.        printf ("# [ %g, %g\n#   %g, %g]\n",  cov00, cov01, cov01, cov11);
63.        printf ("# sqrerr = %g\n", sqrerr);
64.        printf("Nacisnij dowolny klawisz, aby zakonczyc...\n");
65.        getchar();
66.       return 0;
67. }
```

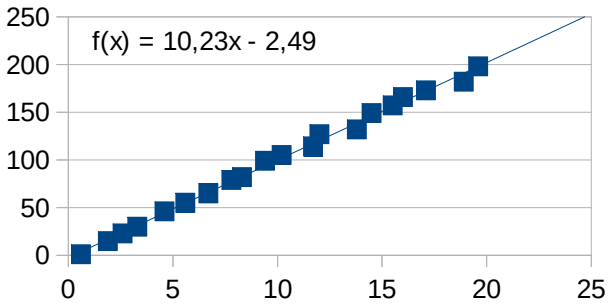
Analizę kodu programu pozostawia się czytelnikowi.

Kompilator produkuje plik uruchomieniowy programu, który skojarzony jest bezpośrednio z

biblioteką dynamiczną *libgsl.dll* oraz *libgslcblas.dll*, a nie jak można było sądzić z biblioteką *libgsl.a*. Biblioteki te można dostarczyć z pliku **gsl-1.8-bin.zip** pobranego z tej samej witryny co plik z bibliotekami statycznymi GSL. Pliki należy rozpakować i umieścić w tym samym katalogu co plik wykonywalny programu **.exe**. Efekt uruchomienia programu widoczny jest niżej:

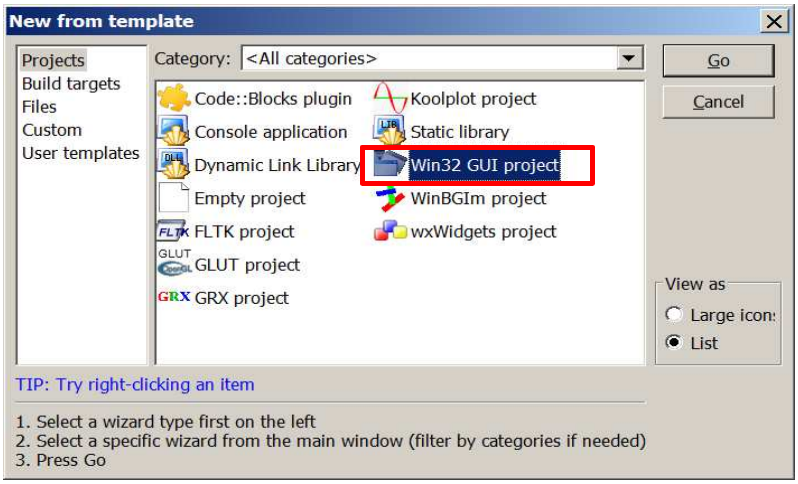


Poza tym w katalogu programu zostanie utworzony plik *daneapprox.csv*, który w celu zobrazowania wygenerowanych danych na wykresie można otworzyć w aplikacji MS Excel lub Open Office Calc. Oto efekt wykreślenia wygenerowanych danych w Open Office Calc z dodaną linią trendu (aproksymacja liniowa) i jej równaniem:

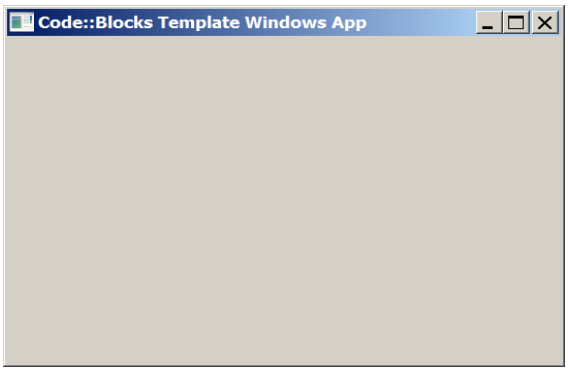


3.3 Programowanie prostej aplikacji okienkowej Windows

Ze względu na to, że kod potrzeby do utworzenia pustego okienka Windows liczy nieco ponad 90 linii - wygenerujemy go automatycznie. W tym przypadku zaczniemy od utworzenia projektu typu Win 32 GUI Project w środowisku Code::Blocks:



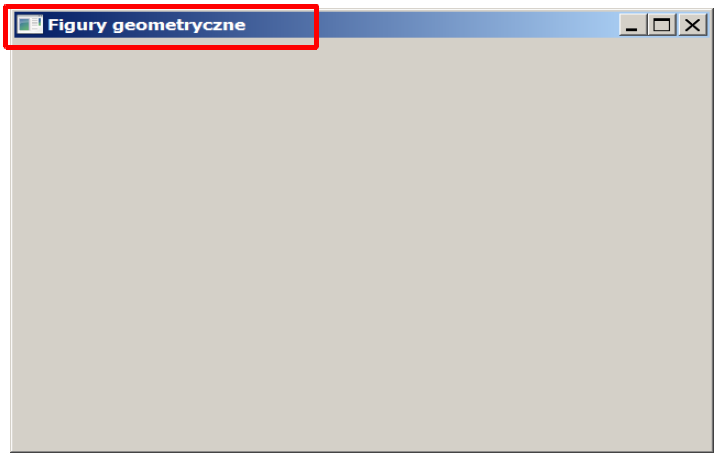
Jako typ projektu aplikacji okienkowej wybieramy *Frame based* (standardowa plkikacja C na bazie Windows API). Dalej nadajemy nazwę projektu i nie wykonujemy modyfikacji opcji domyślnych. Po zakończeniu tworzenia i konfigurowania projektu można go skompilować (Ctrl i F9) a następnie uruchomić (Ctrl i F10). Powinno pojawić się okno przypominające:



Na początek zmienimy nazwę okna wyświetlaną na pasku informacyjnym na "Figury geometryczne". W tym celu należy dokonać edycji łańcucha znaków w wierszu 43:

```
41          0,                  /* Extended possibilites for variation */
42          szClassName,        /* Classname */
43          "Code::Blocks Template Windows App",       /* Title Text */
44          WS_OVERLAPPEDWINDOW, /* default window */
45          CW_USEDEFAULT,      /* Windows decides the position */
46          CW_USEDEFAULT,      /* where the window ends up on the screen */
47          544,                /* The programs width */
48          375,                /* and height in pixels */
```

Zmienimy również rozmiary okna edytując wartości w wierszu 47 i 48. Dokonujemy następujących zmian: 544 na 640 a 375 na 480 (uzyskując rozmiar okna 640x480). Oto rezultat zmian:



Teraz napiszemy kod rysujący figury geometryczne. Kod wykonujący to zadanie umieścimy w definicji funkcji `WindowProcedure()`. W celu umożliwienia rysowania w oknie graficznym zadeklarujemy zmienne:

- `PAINTSTRUCT ps` – struktura zawierająca informacje wykorzystywane przy rysowaniu w oknie graficznym
- `HDC hdc` – uchwyt indeksu urządzenia

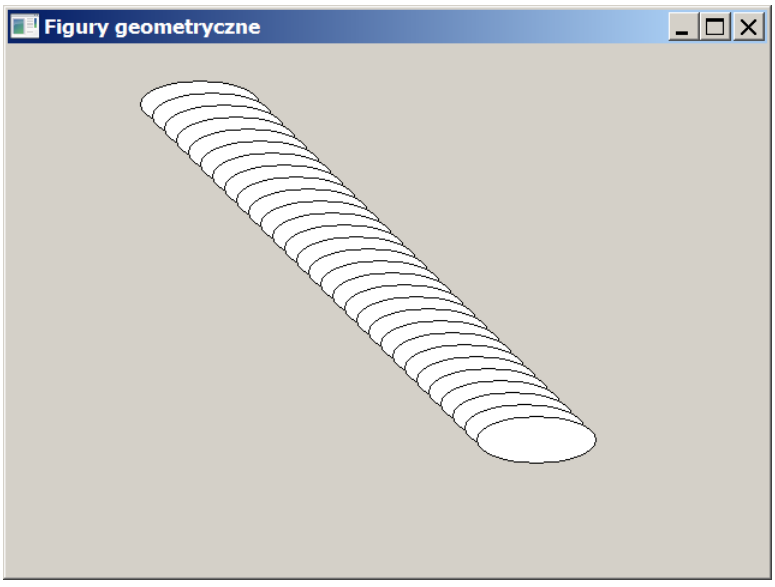
Deklaracje te umieszczamy na początku definicji funkcji WindowProcedure() w linii 76 i 77:

```
74  LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
75  {
76      PAINTSTRUCT ps;
77      HDC hdc;
78      int i;
```

Deklarujemy również zmienną 'i' wykorzystywaną dalej jako licznik pętli. Od wiersza 85, poza blokiem case WM_DESTROY: … break; wpisujemy kod głównej procedury rysującej:

```
case WM_PAINT:
hdc = BeginPaint(hwnd, &ps);//przygotowanie obszaru okna
/*instrukcje obslugi grafiki*/
for(i=1;i<30;i++)//rysowanie 30 elips
        Ellipse(hdc, 10*i+100, 10*i+20, 10*i+200, 10*i+60);
EndPaint(hwnd, &ps);//zakonczenie mozliwosci rysowania
break;
```

Po wykonaniu kompilacji, uruchamiamy program. Rezultat jego działania jest następujący:



Część III: Elementy programowania obiektowego - Artur Opaliński

1. Charakter programowania obiektowego

Jak powiedział Steve Jobs, współzałożyciel firmy Apple: komputer jest najbardziej niezwykłym narzędziem, które stworzył człowiek; jest niczym rower dla naszego umysłu.

Programowanie imperatywne, opisane w poprzednich dwóch częściach skryptu, jest bardzo bliskie umysłowości człowieka. Większość z nas miała lub była młodszym rodzeństwem, i zna z praktyki łatwość rozkazywania, sterowania poprzez wydawanie kolejnych poleceń w celu realizacji zadania krok po kroku. Programowanie imperatywne – wraz z jego zaletami i wadami – stało się także najwcześniejszym sposobem programowania komputerów. Jednak umysł ludzki potrafi znacznie więcej, niż wydawać czy rozumieć rozkazy. Stąd też narzędzie, jakim są języki programowania komputerów, ewoluuje aby umożliwić wyrażenie innych pojęć, których w naturalny sposób używa nasz umysł.

Programowanie – niezależnie od przyjętego języka programowania – pozwala abstrahować od pewnych szczegółów związanych z działaniem komputera. Na przykład programując w C, nie trzeba rozważać momentów, w których dane zostaną przeniesione z pamięci operacyjnej do procesora, czy rozstrzygać w którym rejestrze procesora powinny się znaleźć. Programując w C, nie rozważa się przeważnie nawet tego, w ile i jakich procesorów wyposażono programowany komputer. Widzi się zatem wówczas uproszczony model prawdziwego komputera. Programowanie zawsze oznacza szukanie sposobu na przedstawienie rozwiązania aktualnego problemu, w ramach możliwości uproszczonego modelu komputera, jaki oferuje wybrany język programowania. Tym samym łatwo zgadnąć, że język programowania znacząco wpływa na łatwość, czy wręcz możliwość znalezienia powiązania między modelem komputera a rozwiązywanym problemem.

Idealnym rozwiązaniem byłoby stosowanie takiego modelu komputera, w którym występują wyłącznie pojęcia naturalne dla rozwiązywanego problemu oraz dla ludzkiego umysłu, a nie dla zasad działania maszyny. Znacznie ułatwiałoby to szukanie rozwiązania, gdyż pozwalałoby skupić się na rozwiązywaniu problemu, a nie na ograniczeniach narzędzia. Istnieją języki nakierowane na rozwiązanie pewnych typów problemów: LISP – ułatwiający operacje na listach danych, PROLOG – pozwalający przedstawić każde rozwiązanie jako ciąg decyzji, itp. Języki te jednak tak ściśle narzucają myślenie kategoriami swojego typu problemów, że słabo nadają się do szukania rozwiązań z innych domen.

Programowanie obiektowe ma w zamyśle dostarczyć programiście pojęć, za pomocą których można opisać elementy problemu. Pojęcia te pozostają na tyle ogólne, że nie ograniczają typów rozwiązywanych problemów. Pozostawiają każdorazową możliwość stworzenia elementów dopasowanych do indywidualnego problemu oraz do wybranego sposobu jego rozwiązywania.

Programowanie obiektowe polega na stosowaniu obiektów do reprezentowania elementów rozwiązywanego problemu. Towarzyszy temu zamysł każdorazowego dopasowania tworzonego programu do rozwiązywanego problemu poprzez tworzenie nowych rodzajów obiektów w miarę potrzeby. Innymi słowy, w programowaniu obiektowym opisuje się rozwiązanie w oparciu o pojęcia związane ze sposobem myślenia o problemie, z ludzkim umysłem, a nie bezpośrednio w oparciu o zasady działania komputera. Dzięki temu w kodzie programu wyraźnie uwidaczniają się pojęcia wynikające ze specyfiki problemu i jego rozwiązania. Przez to to tworzenie, a następnie utrzymanie programów wymaga mniej wysiłku.

Obiekty w każdej chwili działania programu znajdują się w pewnym stanie. Można też zlecać im zadania do wykonania. Stanowi to analogie do bardzo wielu pojęć czy przedmiotów świata rzeczywistego, również posiadających pewien stan, na który można wpływać. Z punktu widzenia poprzednich poznanych dotąd metod programowania można wyobrażać siebie, że obiekt zawiera w sobie funkcje, oraz zmienne, na których te funkcje operują. Obiekt łączy więc w sobie dane i operujący na nich kod.

Pięć podstawowych zasad składa się na ideał programowania obiektowego:

1. **Wszystko w programie jest obiektem.**
   Obiekt można sobie wyobrażać jak „sprytną zmienną", która potrafi nie tylko przechowywać dane, ale także samodzielnie wykonywać zlecane zadania
2. **Program stanowi zbiór obiektów.**
   Wykonanie programu polega na tym, że obiekty z tego zbioru zlecają sobie wzajemnie zadania, poprzez przesyłanie komunikatów. Komunikat oznacza w tym wypadku wywołanie określonej funkcji, należącej do obiektu.
3. **Każdy obiekt powstaje na bazie zbioru innych obiektów**.
   Innymi słowy nowe obiekty buduje się, korzystając z istniejących obiektów. Dzięki temu złożony program przekłada się ostatecznie na proste obiekty początkowe.
4. **Obiekty mogą być odmiennych typów.**
   W programowaniu obiektowym mówi się, że każdy obiekt jest instancją pewnej klasy (jest bytem pewnego typu). Klasy różnią się przede wszystkim rodzajami komunikatów, jakie mogą odbierać ich obiekty.
5. **Wszystkie obiekty związane z danym typem mogą odbierać takie same komunikaty.**
   Skoro nowe obiekty powstają na bazie istniejących obiektów, to nowy obiekt potrafi reagować również na komunikaty zdefiniowane dla jego „przodków" . Wykorzystanie takiej możliwości oferuje znaczące korzyści w programowaniu obiektowym.

1.1 Programowanie obiektowe a strukturalne

Pierwsze kontakty z językiem C polegają na zrozumieniu i zapamiętaniu składni pętli, warunków, czy parametrów funkcji. Kolejnym etapem rozwoju jest dostrzeganie w kodzie powtarzalnych fragmentów, wartych wydzielenia w postaci funkcji.

Programowanie obiektowe daje drugą szansę dla osób, które przy wcześniejszych kontaktach z C nie poczuły się pewnie w omawianym materiale. Tworzenie klas dla obiektów przypomina deklarowanie, co będzie w programie, zamiast nacisku na poszczególne instrukcje czy konstrukcje językowe. Oczywiście, ostatecznie każdy obiekt zawiera dane i kod, i ów kod wymaga napisania i przetestowania – ale po sprawnym rozplanowaniu obiektów i klas może to być już dużo prostsze zadanie, stanowiące też już tylko pewną część całego rozwiązania.

Jedną z istotnych różnic w podejściu strukturalnym i obiektowym jest to, że przy podejściu strukturalnym oddzielnie rozważa się kod (funkcje) i dane (struktury do przechowywania danych). Tymczasem w podejściu obiektowym operuje się obiektami, które uzyskują sens dzięki połączeniu w sobie tych dwóch składników.

O programowaniu strukturalnym Niklaus Wirth napisał książkę o znamiennym tytule „*Algorytmy + struktury danych = programy*", podkreślając w niej jak obrany algorytm narzuca sposób przechowywania danych, zaś dobór danych wpływa na algorytmy. Analogiczny tytuł dla programowania obiektowego brzmiałby: „*Obiekty + komunikaty = programy*", gdyż tutaj na kształt programu najistotniej wpływają decyzje dotyczące zaprojektowania obiektu i jego reakcji na

komunikaty.

1.2 Wady i zalety programowania obiektowego

Obiekty w oprogramowaniu stanowią odpowiedniki obiektów realnego świata, czy pojęć stosowanych przez ludzki umysł, co pozwala zmniejszyć złożoność programu i poprawić czytelność jego struktury.

Programowanie obiektowe narzuca czytelną, modularną strukturę programu. Wymaga etapu projektowania, czy choćby przemyślenia szkieletu większej części rozwiązania, co początkujący programiści łatwo pomijają przy programowaniu czysto imperatywnym. Oczywiście taki etap projektowania, choć oddala moment rozpoczęcia kodowania, pozwala lepiej rozplanować pracę i sprzyja tworzeniu lepszego kodu.

W programowaniu obiektowym w naturalny sposób w kolejnych rozważaniach pomija się szczegóły implementacji pomniejszych rozwiązań, bazując już tylko na jasno zdefiniowanych, zewnętrznych cechach obiektów (tzw. interfejsie).

Dzięki podejściu obiektowemu, modularny kod o wyraźnych interfejsach można łatwo modyfikować czy rozwijać: nowe obiekty mogą powstawać dzięki wykorzystaniu istniejących. Łatwo też zastąpić istniejący obiekt jego ulepszoną wersją.

Programowanie obiektowe w bardzo naturalny sposób uwypukla korzyści bibliotek programistycznych, które zawierają stworzone wcześniej komponenty programów, gdyż pozwala łatwo je adaptować czy modyfikować przez programistę, który z nich korzysta.

Wady podejścia obiektowego do programowania wskazywane są rzadziej niż zalety. Z całą pewnością w ramach bardziej zaawansowanych narzędzi rozwiązywania problemów, programowanie obiektowe wprowadza dodatkowe pojęcia: dziedziczenie, przeciążanie, polimorfizm, hermetyzacja. Ich opanowanie wymaga wysiłku, aby móc korzystać z ich zalet. Z kolei intensywne wykorzystywanie tych zaawansowanych metod prowadzi do tworzenia kodu bardzo uogólnionego, abstrakcyjnego, a więc potencjalnie utrudniającego szybką analizę samego kodu, bez korzystania z dodatkowej dokumentacji. Zalety programowania obiektowego uwidaczniają się najpełniej przy pisaniu dużych ilości kodu, na przykład obszernych bibliotek, jednak wykorzystanie tak rozbudowanego kodu wymaga stosownie dużych możliwości obliczeniowych, które nie są dostępne we wszystkich urządzeniach programowalnych. Można także wskazać problemy, których rozwiązanie w sposób obiektowy nie przynosi korzyści w stosunku do podejścia strukturalnego czy czysto imperatywnego.

1.3 Przeznaczenie programowania obiektowego

Programowanie obiektowe wnosi największe korzyści podczas pisania dużych programów. To podejście pozwala ściśle odzwierciedlać strukturę wielkich systemów świata rzeczywistego, i stąd nadaje się do modelowania skomplikowanych systemów o złożonych zachowaniach.

Podział programu na obiekty, które rozwiązują z dużą niezależnością poszczególne pod-problemy nadaje się z kolei doskonale do prac zespołowych: po początkowym podziale problemu, prace poszczególnych zespołów nad przydzielonymi im obiektami pozostają w dużej mierze niezależne od siebie.

Możliwości obiektów można łatwo rozszerzać, poprzez definiowanie własnych klas w oparciu o klasy istniejące. Jest to niezwykle korzystne przy stosowaniu bibliotek programistycznych, które w wypadku programowania strukturalnego nie pozostawiają takiej swobody. Stąd też szeroko stosuje się obiektowe biblioteki do obsługi grafiki.

Zastosowanie programowania obiektowego jest więc szerokie, i łatwiej można wymienić przypadki, gdy nie stosuje się tego podejścia. Są to drobne problemy z dobrym rozwiązaniem czysto imperatywnym, oraz rozwiązania w których kładzie się znaczący nacisk na złożoność obliczeniową przestrzenną lub czasową – np. przy programowaniu urządzeń wbudowanych. Za Joe Armstrongiem można powiedzieć, że żeby zaprogramować obiektowo banana, trzeba najpierw zaprogramować goryla, który trzyma tego banana, oraz dodać resztę dżungli.

2. Programowanie obiektowe jako modelowanie problemu

Obiektowa metoda programowania przeznaczona jest do opisu rozwiązania w sposób blisko związany z problemem, choćby pochodzącym z życia codziennego. Typowy problem związany z realnym światem mógłby dotyczyć serwowania dań w barze szybkiej obsługi (czasami zwanego restauracją *fastfood*). W takim barze (patrz rys. 1) można szybko wyróżnić pierwsze obiekty, które na siebie oddziałują. Przyjmijmy, że bar znajduje się przy dworcu głównym, klientem jest Adam, obsługuje go Beata, zaś jako danie zostanie zakupiony hot-dog.

Jeżeli obiekt *Adam* prześle komunikat do obiektu *Beata*, będzie mógł uzyskać *hot-doga*. Zarówno *Adam* jak i *Beata* posługują się komunikatami, które zmieniają ich stan wewnętrzny – są więc obiektami.

Można się spodziewać, że obiekt *Beata* jest tylko częścią większego obiektu *obsługa_baru_przy_dworcu_głównym*, który składa się w tym lokalu w rzeczywistości z wielu obiektów, na przykład: *Beata* (sprzedawca), *Cezary* (kucharz), *mikrofalówka*. Obiekty te również wymieniają między sobą komunikaty, co w efekcie zapewnia obsłużenie *Adama*. Obiekt *mikrofalówka*, mimo iż nie reprezentuje żywej istoty, ma oczywisty wkład w przygotowanie dania, i również posługuje się komunikatami: wysyła np. dzwonkiem komunikat *„czas podgrzewania minął*”, a odbiera np. komunikat *„otwórz drzwiczki”*, dzięki przyciskom na przednim panelu. Spełnia więc warunki, aby uznać go w tym procesie za pełnoprawny obiekt.

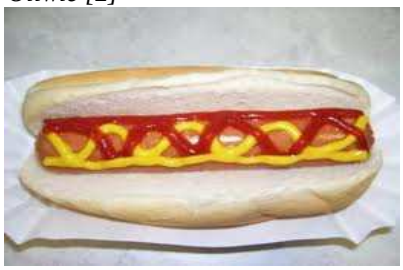
Rysunek 1. Widoczne obiekty baru szybkiej obsługi [1]

Zgodnie z zasadą komponowania obiektów, można wyobrazić sobie również, że cały *bar_przy_dworcu_głównym* jest obiektem, składającym się z innych obiektów, zaś obiekt *Artur* zostaje obsłużony, wchodząc z *barem_przy_dworcu_głównym* w interakcje.

Trudno jednak pominąć fakt, że podane przykłady odnoszą się do obiektów różnych rodzajów. Co więcej, nawet obiekty tego samego rodzaju mogą się różnić: wszak różni *klienci* pojawiają się w barze, choć w tym samym celu i przekazują wówczas podobne komunikaty. Zatem *klient* to pewne ogólne pojęcie (klasa), pod którym mogą kryć się różne osoby (obiekty tej klasy) – niekoniecznie zawsze *Adam*. Mimo to różni *klienci* w zakresie interakcji z barem zachowują się tak samo, wysyłają takie same komunikaty, co oznacza, że są obiektami jednej klasy. Analogicznie można odróżniać np. klasę hot-dogów czy barów, od każdego konkretnego hot-doga czy baru pod konkretnym adresem (rys.2).

a) Bar w Oliwie [2]

b) Bar przy dworcu w Kijowie [3]



c) Hot-dog Adama

d) Hot-dog autora

Rysunek 2. Różne obiekty tych samych klas.

W ogólnym przypadku można powiedzieć: dla każdego przedmiotu w rzeczywistym świecie istnieje, lub da się wymyślić stosowna abstrakcyjna koncepcja, która opisuje:

- oddziaływania między nim, a jego środowiskiem,
- jego wewnętrzną organizację, czy wewnętrzne działanie.

W programowaniu obiektowym taki przedmiot nazywany jest instancją (bytem) pewnej klasy, i jest obiektem. Abstrakcyjna koncepcja która opisuje jego cechy – to właśnie jego klasa.

Również mniej namacalne pojęcia, czy wręcz pojęcia całkiem abstrakcyjne które człowiek wykorzystuje w rozumowaniu (figury geometryczne, wyobrażenia) można postrzegać jako obiekty i przyporządkowywać je do klas.

Z każdym obiektem związane są:

- jego stan,
- jego możliwe zachowania,
- jego identyfikator.

Zachowania obiektu można z technicznego punktu widzenia postrzegać jako operacje, które dany obiekt jest w stanie wykonywać. Z kolei stan obiektu jest wynikiem wykonania przez niego operacji, a więc przechowuje efekt tych operacji. Każdy obiekt ma też przypisany identyfikator, czyli nazwę która go jednoznacznie identyfikuje.

W przykładzie z barem szybkiej obsługi, można przewidywać, że obiekt klasy *sprzedawca* ma identyfikator *Beata*, że może znajdować się w jednym z dwóch stanów – albo przyjmuje zamówienie, albo realizuje zamówienie, stosownie do jego treści. Obiekt wykazuje też następujące zachowania: odbiera treść zamówienia, podaje (danie, bądź półprodukt), nastawia toster, odbiera danie z tostera, inkasuje należność.

3. Modelowanie baru szybkiej obsługi

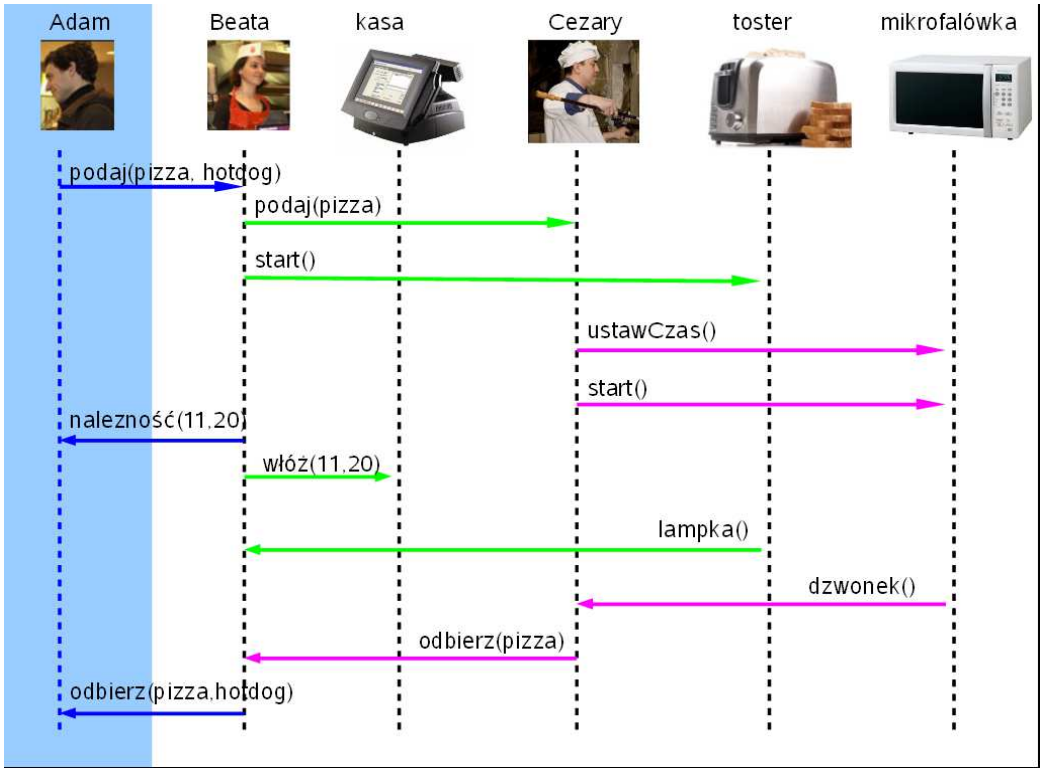
Interakcje w barze szybkiej obsługi można przedstawić formalnie w postaci diagramu (rys. 3).

Rozpoczyna instancja *Adam* klasy *klient*, wysyłając do instancji *Beata* klasy *sprzedawca* prośbę o pizzę i hot-doga. Przygotowaniem pizzy zajmuje się *Cezary* klasy *kucharz*, zaś zrobienie hot-doga polega na podgrzaniu składników w *tosterze*. *Beata* wysyła komunikat do *Cezarego*, aby podał pizzę, oraz komunikat do *tostera*, polegający na włączeniu grzania. Zanim dania będą gotowe, *Beata* wysyła do *Adama* komunikat odnośnie zapłacenia należności.

Tymczasem *Cezary* ustawia w *mikrofalówce* czas odpowiedni do dania i włącza ją. Gdy podgrzewanie jest zakończone, *toster* wysyła poprzez zaświecenie lampki komunikat do *Beaty*. *Mikrofalówka* wysyła dzwonkiem komunikat do *Cezarego*, potwierdzając gotowość dania. *Cezary* wysyła do *Beaty* komunikat, aby odebrała pizzę. *Beata* – posiadając już oba gotowe dania – wysyła do *Adama* komunikat, żeby je od niej odebrał.

Przykład zwraca uwagę na ważne cechy programowania obiektowego:

- Za pomocą obiektów i ich interakcji można opisywać złożony świat czytelniej, niż za pośrednictwem algorytmów programowania imperatywnego. Opis powyższych interakcji w sposób strukturalny byłby skomplikowany, jeżeli miałby uwzględniać, że czas i kolejność wykonywania czynność są uzależnione od innych zdarzeń.
- Hermetyzacja – *Adam*, aby móc skorzystać z czynności wykonywanych przez *bar_przy_dworcu_głównym* wcale nie musi komunikować się ze wszystkimi zawartymi w barze obiektami. Co więcej, nawet nie musi mieć świadomości, jak jest zorganizowana praca w barze, albo jak obsługuje się urządzenia. Dzięki temu nie dotyczą go też zmiany obiektu *bar_przy_dworcu_głównym* – wyposażenia czy recept kuchennych. Dla klienta wystarcza – i musi wystarczyć – znajomość zachowań w obszarze oznaczonym na diagramie (rys. 3) kolorem niebieskim. Również w programowaniu obiektowym ukrywa się część możliwości obiektu poprzez mechanizm zwany hermetyzacją lub enkapsulacją. Dzięki temu korzystanie z obiektu staje się prostsze, zaś ewentualne zmiany w jego organizacji wewnętrznej stają się niewidoczne z zewnątrz.



Rysunek 3. Diagram interakcji między obiektami w barze szybkiej obsługi

- Klasy – skoro *bar_przy_dworcu_głównym* jest tylko jedną instancją klasy *bar_szybkiej_obsługi*, oznacza to dla *Adama*, że w identyczny sposób jest w stanie skorzystać ze wszystkich innych barów tej klasy: wszystkie one będą wykorzystywały te same komunikaty: *podaj*, *naleznosc*, *odbierz*. W programowaniu jest to także zaleta – każda powtarzalność daje się naturalnie i bez wysiłku wykorzystać.
- Dwa punkty widzenia – żeby zamodelować interakcje w barze, korzystając z wymienionych wyżej obiektów, wystarczy wykorzystać wymianę komunikatów. Jest to punkt widzenia programisty-użytkownika klas. Innym punktem widzenie jest jednak spojrzenie programisty-twórcy klas, który wcześniej musiał dogłębnie przemyśleć i stworzyć nie tylko elementy dotyczące komunikatów, ale i zaprogramować wszystkie wewnętrzne algorytmy, wszystkich tych obiektów. Taki dualizm spojrzeń, rozróżnianie tego, co w środku, i tego, co dostępne z zewnątrz jest istotnym elementem programowania obiektowego i wiąże się ze stosowaniem ról programisty-twórcy klas i programisty-użytkownika klas.

4. Modelowanie stosu

Pojęcie stosu

W informatyce, oprócz pojęcia prostych zmiennych, na przykład typu *int* czy *float*, stosuje się zmienne złożone, na przykład tablice, listy, struktury. Jedną ze złożonych struktur danych jest stos. Stos, podobnie jak tablica, pozwala przechowywać wiele wartości.

Jak wiadomo, w tablicy można jednakowo łatwo uzyskać dostęp do dowolnego elementu. W przykładowym programie (przykład 1) zadeklarowano (linia 7) tablicę *tab* zawierającą N elementów typy *float*. Stałą N zdefiniowano uprzednio jako 10 (linia 4). Poczynając od ostatniego elementu, wszystkie elementy tablicy zostały wypełnione wartościami (linie 10, 11). Użyto do tego celu funkcji *sin*(), co wymagało dołączenia pliku nagłówkowego math.h (linia 2). Następnie (linie 13,14) wydrukowano co drugi element tablicy, poczynając od pierwszego elementu, i uwzględniając, że elementy w tablicach w C i C++ indeksowane są od zera. Wykorzystanie do wydruku funkcji *printf*() wymagało dołączenia pliku nagłówkowego stdio.h (linia 1). Jak widać zatem, do elementów tablicy można sięgać w dowolnej kolejności: od końca, od początku, czy nawet przeskakując nieistotne elementy (w przykładzie drukuje się co drugą wartość); jednakowo swobodnie można dokonywać zarówno odczytów, jak i zapisów.

Przykład 1. Dostęp do elementów tablicy

```
 1.      #include <stdio.h>
 2.      #include <math.h>
 3.
 4.      #define N 10
 5.
 6.      int main(){
 7.      float tab[N];
 8.      int i;
 9.
10.      for(i=N-1; i>=0; i—)
11.          tab[i]=sin(i);
12.
13.      for(i=0; i<N; i=i+2)
14.          printf("Na pozycji %d jest wartosc %f\n",i,tab[i]);
15.
16.      getchar();
17.      }
```

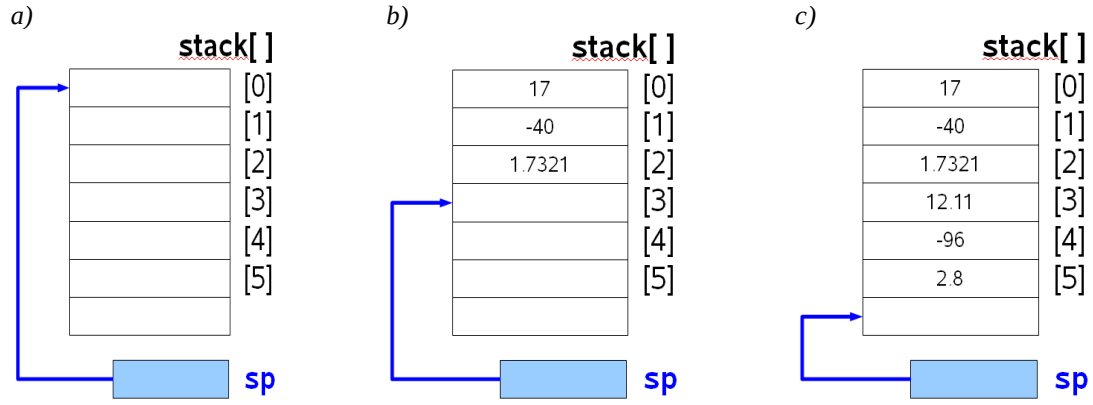
W przeciwieństwie do tablicy, struktura danych zwana stosem pozwala dopisywać i odczytywać dane tylko na wierzchołku stosu. Mimo to pozwala przechowywać wiele danych. Przez analogię można wyobrazić sobie stos delikatnych, porcelanowych talerzy, który również przechowuje dużo elementów. Można bezpiecznie dołożyć kolejny talerz tylko na wierzchołku stosu, lub pobrać go z wierzchołka stosu; nie da się jednak wcisnąć talerza na spód stosu czy w jego środek; nie da się także wyszarpnąć talerza ze środka czy z dna stosu. Podsumowując, można powiedzieć, że elementy można pobierać ze stosu wyłącznie w kolejności odwrotnej do tej, w jakiej były

dodawane. Zarówno stos talerzy, jak i stos w pamięci komputera nie może rosnąć w nieskończoność ze względu na ograniczenia wytrzymałościowe (talerze) lub pojemność pamięci (komputer), niemniej w rozważaniach uwzględnia się działanie stosu nie prowadzące do jego przepełnienia (*overflow*).

Struktura danych zwana stosem, ograniczona dwiema operacjami: odłóż na stos (*push*), pobierz ze stosu (*pop*), wydaje się dużo mniej wygodna niż tablica. Pozwala jednak w bardzo łatwy sposób rozwiązywać niektóre problemy, a nawet została opatentowana. Znajomość tej struktury na przykład bardzo ułatwia zaprogramowania kalkulatora z odwrotną notacją polską. Tymczasem jednak stos posłuży za przykład obiektu, który można modelować metodami programowania obiektowego.

W programowaniu stos uzupełniony jest wskaźnikiem (*stack pointer, sp*), który wskazuje wierzchołek stosu. Dodatkowo przyjęło się, że stos przyrasta w dół, to znaczy że pierwszy odłożony element trafia pod najwyższy dostępny adres pamięci, zaś kolejno odkładane elementy trafiają pod kolejne, niższe adresy. Każdorazowo odłożenie elementu na stos lub pobranie elementu ze stosu powoduje dodatkowo aktualizację wskaźnika stosu, tak aby zawsze wskazywał na aktualny wierzchołek stosu.

Działanie stopniowo zapełniającego się stosu przedstawiono na rys. 4: stos początkowo jest pusty (rys. 4a), następnie (rys. 4b) w wyniku kilku operacji *push* został częściowo zapełniony; być może nastąpiły tymczasem także pewne operacje *pop* – w każdym razie, ostatecznie na stosie pozostają 3 wartości: 17, -14, oraz 1.7321, które na pewno odkładane były w takiej właśnie kolejności. Rys. 4c przedstawia sytuację, w której stos został zapełniony całkowicie – być może tylko sześcioma operacjami *push*, ale mogło ich być więcej, jeżeli tymczasem wykonywano również operacje *pop*. Na pewno wartość 17 została odłożona wcześniej niż wszystkie pozostałe widoczne wartości, zaś wartość 2.8 – na pewno jako ostatnia.



Rysunek 4. Stos w różnych stanach:
a) stos pusty; b) stos częściowo zapełniony; c) stos w całości zapełniony (overflow)

4.1 Modelowanie stosu w programie

Żeby móc korzystać z zalet stosu w programie, należy go zaprogramować. W tym celu trzeba dostrzec, że stos jest obiektem:

- można wysyłać do niego komunikaty (*push* i *pop*), na które on reaguje, wykonując stosowne operacje;
- stan, w jakim znajduje się stos, jest opisany przez *sp*. Co więcej, nie potrzeba żadnego innego atrybutu, aby go opisać – jedyną istotną cechą jest jego zapełnienie; wartości

zapisanych na stosie danych nie wpływają wcale na jego działanie a zatem nie opisują jego stanu;

- można wyobrazić sobie, że rys. 4. przedstawia 3 różne stosy, zapełnione w różnym stopniu – w takim razie naturalne jest nazwanie ich (nadanie identyfikatorów).

Podstawowe informacje o stosie, zapisane w języku C++, mają postać przedstawioną w przykładzie 2.

Przykład 2. Model stosu w języku C++

```
1.   double stack[6];
2.   int sp = 0;
3.
4.   void push(double v) {
5.        //kod pominięty
6.   }
7.
8.   double pop(void) {
9.        //kod pominięty
10.  }
```

W liniach 1 i 2 zostają zadeklarowane zmienne związane ze stosem. Sam stos to jednowymiarowe tablica *stack* zawierająca 6 elementów typu *double;* zmienna *sp* to wskaźnik stosu, wskazująca najpierw, w pustym stosie, na jego początkowy element,

W liniach 4-6 zadeklarowana jest funkcja realizująca zachowanie o nazwie *push*. Potrzebuje jednego parametru, jakim jest wartość typu *double*, którą należy odłożyć na stos. Ciało tej funkcji nie jest na razie istotne; wiadomo że ma zapewnić zachowanie *push.*

W liniach 8-10 zadeklarowana jest funkcja realizująca zachowanie o nazwie *pop*. Zwraca pojedynczą wartość typu *double*, którą pobrała ze stosu. Ciało tej funkcji również nie jest na razie istotne.

Widać już jednak, że stosując zaledwie dwie zmienne, oraz dwie funkcje, można utworzyć model pojęcia (stos), związanego z przechowywaniem danych, traktowanego jak obiekt wykonujący polecenia. Wydawanie poleceń, czyli przesyłanie komunikatów, odbywa się poprzez wywołanie stosownej funkcji (*push()* albo *pop()*). Aby skorzystać z usług takiego obiektu do przechowywania danych, nie ma w ogóle potrzeby używania zmiennych *stack[ ]* czy *sp* – a mimo to można z jego pomocą zachowywać i odczytywać dane.

Już tak prosty przykład przybliża dwie zalety – i jedna wadę – programowania obiektowego:

- Pojęcie klas – skoro każdy obiekt jest zaledwie instancją swojej klasy, to po jednokrotnym zdefiniowaniu klasy opisującej *stos* – można w programie utworzyć i używać dowolnie dużo obiektów tej klasy. Nie trzeba w tym celu powielać raz napisanych definicji, co byłoby nieuniknione przy programowaniu strukturalnym.
- Interfejs – do wykorzystania tak zaprogramowanego stosu wystarcza wywoływanie push() i pop(). Nie ma potrzeby odwołań do stack[] czy sp. Tym samym interfejs, czyli linia styku, na której następować będzie komunikacja reszty programu z instancjami klasy stos, jest bardzo jasno określona. Korzystanie z obiektu staje się prostsze. Programista korzystający z takiego obiektu nie musi się zastanawiać, czy dane przechowywane są w tablicy, a jeśli tak, to czy przyrastają od góry czy od dołu. Skupia się na tylko dwóch funkcjach, które oferują mu wszystko, czego potrzebuje od stosu. W

związku z tym programując inne obiekty, które korzystają z klasy *stos*, nawet nie wiadomo, do której tablicy trafiają dane, ani czy w rzeczywistości ostatecznie dane trafiają do tablicy. Idąc dalej, jeżeli z czasem klasa *stos* zostanie zmodyfikowana, i nie będzie przechowywać danych w tablicy, lecz np. na liście, albo w pliku dyskowym – sposób korzystania z obiektów tej klasy w ogóle się nie zmieni: nadal wywoływać się będzie tylko *push()* i *pop()*, w ten sam sposób. Tym samym nie będzie konieczne modyfikowanie żadnej części programu, która korzysta z tego obiektu.

- Za wadę można traktować wskazanie, że przecież żeby przechować dane w jednowymiarowej tablicy nie trzeba na wstępie tyle programować. Rzeczywiście, jednak dzięki programowaniu uzyskujemy tak naprawdę nowy typ danych, o innych cechach niż tablica, który przydaje się do innych zastosowań. Niestety, wykorzystanie zalet obiektowości i późniejsze łatwiejsze programowanie dzieje się kosztem wysiłku włożonego w zrozumienie nowych pojęć.

5. Elementy programowania obiektowego w języku C++

Programowanie obiektowe wprowadza wiele pojęć nowych w stosunku do programowania strukturalnego. Bardziej zaawansowane pojęcia, takie jak przeciążanie, wzorce (*templates*), metody wirtualne, wyjątki czy programowanie współbieżne, nie zostaną tu omówione.

5. 1 Deklarowanie klas

Klasę deklaruje się – jak widać na przykładzie 3 – z użyciem słowa kluczowego *class*. Każda klasa musi mieć nazwę. Klasa może być pusta, ale nawet wtedy musi wystąpić para nawiasów klamrowych: ***{*** oraz ***}*** , oraz znak średnika ***;*** kończący deklarację.

Przykład 3. Podstawy składni deklaracji klasy

```
1.    class Klient {
2.          // klasa pusta
3.    };
```

Deklaracja klasy opisuje charakter wszystkich obiektów (bytów, instancji) tej klasy, w tym ich zachowania. Elementami klasy są funkcje, które w tym wypadku nazywa się *metodami*. Elementami klasy są także zmienne, nazywane *polami danych*. Z całą pewnością klasa opisująca klientów baru (przykład 4) powinna opisywać przynajmniej dwie wymienione na diagramie (rys. 3) metody: *odbierz*(), oraz *naleznosc*().

Przykład 4. Elementy deklaracji klasy

```
1.  class Klient {
2.      void odbierz(char danie[]);
3.      void naleznosc(float kwota);
4.      float pieniadze;
5.      int glod;
6.  };
```

Załóżmy, że metoda *danie*() wymaga jednego parametru, jakim jest łańcuch znakowy opisujący nazwę dania (linia 2), zaś metoda *naleznosc*() potrzebuje parametru typu rzeczywistego, opisującego kwotę do zapłacenia (linia 3).

Metody zadeklarowane są tak, jak deklaruje się funkcje: poprzez podanie ich prototypu; jedyną różnicą jest fakt, że prototypy metod podane są w ramach deklaracji klasy, a nie poza nią. Na tym etapie programowania podawanie zawartości metod (definicja ich wnętrza) nie jest jeszcze konieczna, choć możliwa. W programowaniu obiektowym wyraźnie rozdziela się dwa pojęcia deklaracji i definicji, stosowane także w programowaniu strukturalnym. Deklaracja klasy polega tylko na wymienieniu jej składowych: ich nazw, typów, parametrów metod. Definicje – czyli ostateczne opisanie treści metod – tworzy się zazwyczaj oddzielnie, i nieco później. Pozwala to oddzielić etap planowania, projektowania rozwiązania, czyli myślenia o problemie, od etapu kodowania – czyli od myślenia o składni języka i szczegółach algorytmów.

Deklaracja klasy oprócz metod zawiera także deklaracje pól danych. Opisują one stan wewnętrzny obiektu: w wyniku uzyskania komunikatu *naleznosc* będą na pewno malały zasoby finansowe klienta (pole *pieniadze*, zadeklarowane w linii 4). Z kolei głód (linia 5) powinien maleć podczas reakcji na komunikat *odbierz*.

5.2 Hermetyzacja

Jednak diagram (rys. 3) nie ujmuje takich cech klienta jak pieniądze czy głód. W realnym świecie są to prywatne informacje klienta, nieznane dla nikogo z obsługi baru. Programowanie obiektowe pozwala uwzględnić również ten aspekt rzeczywistości: deklarację klasy można podzielić na *sekcje public*, *private* lub *protected*. Deklaracje umieszczone w sekcji *public* są jedynymi elementami, z których mogą korzystać obiekty odmiennych klas. Deklaracje można też umieszczać w sekcji *private* – są wówczas dostępne tylko dla metod tej samej klasy, a niedostępne dla metod należących do innych klas. Sekcja *protected* zostanie omówiona dalej.

Deklarację klasy dzieli się na sekcje, podając słowo kluczowe *private*, *public* lub *protected*, zakończone znakiem dwukropka **:** (przykład 5). Nazwę sekcji można podawać wielokrotnie w ramach jednej deklaracji klasy, czyli można np. zadeklarować najpierw pewne elementy prywatne, potem część elementów publicznych, potem znów prywatne, itd. Jeżeli nazwa sekcji nie jest podana, to domyślnie elementy klasy należą do sekcji *private*; zatem przykład 4 nie odpowiada do końca diagramowi z rys. 3, gdyż w przykładzie tym wszystkie elementy domyślnie należą do sekcji prywatnej, co powoduje że metody *naleznosc*() czy *odbierz*() nie mogłyby być wywoływane przez obiekty należące do klasy obsługi baru.

Przykład 5. Sekcje w deklaracji klasy

```
 1.   class Klient {
 2.     public:
 3.       void naleznosc(float kwota);
 4.       void odbierz(char danie[]);
 5.       void zjedz(void);
 6.     private:
 7.       float pieniadze;
 8.       int glod;
 9.       void skorzystaj_z_baru(void);
10.   };
```

Podział na sekcje nie istnieje jednak tylko w celu większego upodobnienia modelu do rzeczywistości. Realnym i częstym problemem programowania są ludzkie błędy, popełnianie przez programistę. Stosowanie sekcji prywatnej pozwala ukryć przed programistą-użytkownikiem klasy te elementy, z których nie ma potrzeby korzystać. Skoro programista-twórca klasy zaoferował publiczne metodę *naleznosc*() - która będzie zmniejszać ilość pieniędzy, oraz *odbierz*() - która będzie prowadzić do zaspokajania głodu – to zmienne opisujące głód i zasoby finansowe nie muszą być już udostępnianie. Nie wynika to z faktu ochrony czyjejś prywatności, lecz ma na celu zmniejszenie liczby elementów, którymi operuje programista-użytkownik klasy. Skoro elementów widocznych jest mniej – to programista-użytkownik klas łatwiej może je zapamiętać i opanować posługiwanie się nimi. Powoduje to mniej problemów przy programowaniu. Skoro pewne elementy są niewidoczne – nie można ich użyć błędnie. To znów mniej problemów przy programowaniu. Co więcej, jeżeli programista-twórca klas dostarczy z czasem nieco zmienioną klasę *Klient*, w której np. głód będzie przechowywany w polu typu rzeczywistego zamiast całkowitego, zaś zasoby finansowe w dwóch oddzielnych polach odpowiadających dwóm rodzajom walut – i stosownie do tego zmieni wewnętrzne działanie publicznych metod *naleznosc*() i *odbierz*() – to programista-użytkownik klas nie będzie musiał zmieniać nic w swojej części programu. Mniej problemów przy programowaniu. Z punktu widzenia programisty-użytkownika klas, instancje *Klient* nadal będą zaspokajały głód i wydawały pieniądze – tak jak wcześniej.

Przykład 5 uzupełnia deklarację o dwie dodatkowe metody. Wywołanie publicznej (linia 5) metody *zjedz*() pozwala z zewnątrz zaktywizować obiekt, zainicjować jego dalsze działania w celu zaspokojenia głodu. Następnie metoda *skorzystaj_z_baru*() zostanie wykonana jako reakcja na zachętę do jedzenia (linia 9) – i tym samym spowoduje rozpoczęcie interakcji klienta z barem. Będzie wywołana z metody *zjedz*() - metody z tej samej klasy – czyli nie musi być publiczna, i w celu zmniejszenia możliwości błędów została ukryta w sekcji *private*.

Można też tłumaczyć sobie jej ukrycie tak, że obiekt klasy Klient może poprawnie zareagować na komunikat *zjedz* w dowolny sposób, niekoniecznie korzystając z baru, ale też wykorzystując dowolną inną metodę zależną od swojego stanu wewnętrznego i prywatnych możliwości: *zjedz_kanapki*(), *popros_mame*(), itp. - i te prywatne możliwości są dla programisty-użytkownika klasy nieistotne, więc nie powinny być dla niego widoczne.

Świadomy, celowy dobór elementów deklaracji klasy do sekcji *private* nazywa się *hermetyzacją* (*encapsulation*).

5.3 Tworzenie obiektów

Klasa opisuje potencjalną zawartość swoich instancji, które jednak nie powstają razem z deklaracją klasy. Każda klasa może mieć wiele instancji, dlatego tworzenie *(instatiation)* obiektów jest czynnością oddzielną od deklaracji klasy.

Najprostszy sposób utworzenia obiektu *Adam* klasy Klient pokazano na przykładzie 6 (linia 6). Jak widać, bardzo przypomina on deklarację zmiennej, znaną z pierwszych kontaktów z językiem C:

```
int a;
float b;
```

i mającej ogólnie postać:

```
<nazwa typu> <nazwa zmiennej>;
```

Przyjęto w kodzie nazewnictwo promowane przez Microsoft, choć niezgodne z tradycją języków C i C++, aby klasy nazywać, poczynając od wielkiej litery, zaś nazwy obiektów i zmiennych rozpoczynać małą literą.

Przykład 6. Tworzenie obiektu

```
1.  class Klient {
2.      //znana deklaracja pominięta dla skrócenia przykładu
3.  };
4.
5.  void tworzenie(){
6.  Klient adam;
7.  Klient *wskKlient;  //wskKlient to wskaźnik do obiektów klasy Klient
8.  wskKlient = new(Klient);    //wskKlient wskazuje na nowo utworzoną instancję klasy Klient
9.  Klient *adrian=new(Klient); //adrian to wskaźnik; jest od razu inicjalizowany
10. // reszta kodu pominięta
11. }
```

W języku C istnieje grupa zdefiniowanych przez standard typów (*char, int, long int, float, double*, itp) oraz możliwość tworzenia nowych typów i nazw z użyciem słowa kluczowego *typedef*. Typ opisuje, ile miejsca w pamięci zajmie wartość, oraz w jakim formacie przechowywać bity, reprezentujące wartość. Jednak typ to nie jest zmienna. Typ to tylko ulotne pojęcie, w którym nie da się przechować wartości. Dlatego istnieje również pojęcie zmiennej: to już jest nazwane miejsce w pamięci, w którym można zapisywać wartości, i dzieje się to zgodnie z zasadami właściwymi dla typu tej zmiennej. Mawia się, że zmienna jest pewnym bytem (*instancją*) danego typu.

Klasa jest podobna do typu, gdyż opisuje tylko cechy potencjalnych, jeszcze nieistniejących instancji. Dopiero zadeklarowanie obiektu (*instancji*) danej klasy powoduje, że w pamięci zostaje przydzielone miejsce, które zajmują następnie elementy opisane deklaracją klasy. Dlatego najprostsza metoda tworzenia obiektów jest taka sama, jak metoda tworzenia zmiennych. Wymagania i skutki takiego tworzenia obiektów są takie same, jak tworzenie zmiennych:

- aby zadeklarować obiekt danej klasy, ta klasa musi już być znana, a więc zadeklarowana wcześniej w kodzie (w przykładzie klasa zdefiniowana jest w liniach 1-3, a użyta dopiero w linii 6). Poszczególne metody klasy nie muszą jeszcze być do tego miejsca zdefiniowane; do kompilacji wystarczają ich deklaracje, zawarte w deklaracji klasy.
- obiekt będzie widoczny w tym samym zakresie, co tak samo zadeklarowana zmienna, czyli lokalnie lub globalnie. W przykładzie 6 obiekt *Adam* jest zadeklarowany lokalnie w funkcji

tworzenie(), można więc go używać tylko w tej funkcji, poczynając od linii 6 – nie jest widoczny poza tą funkcją, tak samo jak nie są widoczne zmienne lokalne.

- czas życia obiektu będzie taki sam, jak czas życia zmiennej zadeklarowanej z taką samą klasą. W przykładzie 6 obiekt *Adam* jest zadeklarowany jak zmienna automatyczna (zmienna klasy auto) więc będzie istniał tylko tak długo, jak długo działa funkcja, w której został zadeklarowany. Innymi słowy obiekt zostanie umieszczony w tym samym segmencie pamięci, co tak zadeklarowana zmienna.

Drugą metodą tworzenia instancji klasy jest jej dynamiczna alokacja. W języku C do dynamicznej alokacji i dealokacji pamięci służą między innymi funkcje *malloc*() oraz *free*(). W języku C++ na potrzeby dynamicznej alokacji pamięci dla obiektów stworzono funkcje *new*() i *delete*(), które działają bardzo podobnie, ale dodatkowo uruchamiają konstruktor klasy (przeznaczenie i przydatność konstruktorów zostaną omówione dalej).

W przykładzie 6 w liniach 7-8 utworzono dynamicznie dwa dodatkowe obiekty klasy *Klient*: najpierw w linii 7 zadeklarowano zmienną wskaźnikową *wskKlient*, która będzie służyć do przechowywania adresów do pamięci, w której znajdują się obiekty klasy *Klient*. Jest to zwykła zmienna lokalna, mieszcząca w sobie numer komórki pamięci (liczbę całkowitą) pod którą znajduje się obiekt klasy *Klient.* W linii 8 zmienna *wskKlient* została zainicjalizowana: wywołano funkcję *new*(), która zażądała od systemu przydzielenia pamięci w ilości potrzebnej na jeden obiekt klasy *Klient*, i utworzenie tam takiego obiektu. Funkcja *new()* zwraca uzyskany adres, który zostaje następnie wpisany do zmiennej *wskKlient*. W razie niepowodzenia – czyli np. braku wolnej pamięci – *new()* zwraca specjalną wartość NULL, która oznacza, że obiektu nie udało się założyć.

Deklarację zmiennej wskaźnikowej oraz jej inicjalizację z użyciem *new*() można połączyć w jednej linii kodu, co przedstawiono w linii 9 przykładu 6. Zmienna wskaźnikowa *adrian* została nazwana męskim imieniem rozpoczynającym się literą ***a***, co ułatwia w dalszym kodzie skojarzenie, że odnosi się do obiektu klasy Klient. Warto jednak zauważyć, że w porównaniu z linią 6, gdzie powstawał obiekt *adam*, tu powstały obiekt nie ma nazwy – nazwę ma tylko zmienna wskaźnikowa *adrian*, przechowująca jego adres. Korzystanie ze zmiennej wskaźnikowej wymagać będzie dalej w kodzie nieco innej składni, niż bezpośrednie korzystanie z obiektu.

Oczywiście, choć czytelność kodu jest kluczowa, gdy pragnie się szybciej i łatwiej tworzyć programy, to z punktu widzenie języka czy kompilatora nazwa obiektu albo zmiennej wskaźnikowej do obiektu mogłaby być dowolna, jak długo spełnia wymagania niepowtarzalnego identyfikatora.

Reasumując:

- w linii 6 powstaje instancja klasy Klient, i owa instancja nazywa się *adam;*
- w linii 8 powstaje instancja klasy Klient. Jednak tu instancja nie została nazwana; wskazuje na nią zmienna *wskKlient;*
- w linii 8 powstaje instancja klasy Klient. Tu również instancja nie została nazwana; wskazuje na nią zmienna *adrian*, która mimo podobieństwa nazwy do *adam* jest tylko zmienną wskazującą na nienazwany obiekt, a nie samym obiektem.

Choć sposób identyfikowania trzech powyższych obiektów jest odmienny, to wszystkie one powstały w dokładnie takim samym kształcie, opisanym deklaracją klasy. Każdy z nich oferuje identyczne metody, które realizują ramach każdego obiektu dokładnie takie same zadania, oraz oferuje pola danych o identycznych nazwach i typach. Początkowo – w chwili powstania – pola danych w każdym z tych obiektów inicjalizowane są tak samo. Od momentu powstania każdy obiekt może jednak żyć oddzielnym życiem: jego metody mogą być wywoływane w innych momentach, w innej kolejności i z innymi parametrami, niż metody pozostałych obiektów tej klasy. Prowadzi to z czasem do tego, że pola danych każdego z obiektów przyjmują z czasem odmienne

wartości. Odzwierciedla to odmienny stan wewnętrzny każdego z obiektów: niektóre mogą stać głodne, inne syte; niektóre mogą być zasobne w pieniądze, inne – biedne.

Zmienne *wskKlient* oraz *adrian* to w przykładzie 6 najbardziej typowe, automatyczne zmienne lokalne. Ich zasięg i czas życia są takie same, jak wszystkich automatycznych zmiennych lokalnych, gdyż zostają umieszczone w segmencie stosu procesu. Jednak wskazywane przez nie obiekty, umieszczone zostały w dynamicznie zaalokowanej pamięci w segmencie sterty (*heap*) procesu. Wszystkie obszary pamięci zaalokowane dynamicznie pochodzą ze sterty, i w związku z tym wszystkie istnieją przez cały czas trwania procesu, o ile nie zostaną wcześniej jawnie usunięte funkcją *delete*(). Oznacza to, że choć instancja *adam*, wraz ze swoimi niepowtarzalnymi wartościami pól danych, zniknie gdy skończy działanie funkcja, w której została zadeklarowana – to pozostałe dwa obiekty będą istniały dalej. Trzeba jedynie zadbać w programie, aby ich adresy przekazać do innych funkcji, które będą mogły dalej z tych obiektów korzystać. Jeżeli się tego nie zrobi – obiekty staną się niedostępne: nawet jeżeli wciąż będą zawarte w pamięci, nie będzie już informacji, gdzie je znaleźć.

5.4 Implementacja metod

Deklaracja klasy nie opisuje jeszcze wszystkiego: wymienia się w niej metody, ale nie jest jasne, jak te metody mają działać. Po deklaracji w klasie potrzebna jest dodatkowo definicja zawartości metod: opisanie, co dokładnie, krok po kroku, mają robić. W programowaniu obiektowym definicję metod nazywa się często ich *implementacją*.

Z punktu widzenia deklaracji klasy, przewidziano dwa sposoby definiowania metod:

- implementację *inline*, zwaną także lokalną, która następuje natychmiast po deklaracji metody, czyli jeszcze wewnątrz deklaracji klasy;
- implementację globalną, podawaną po zakończeniu deklaracji klasy. Aby zaznaczyć że definiowany właśnie kod jest uzupełnieniem klasy, poprzedza się nazwę metody nazwą klasy i operatorem złożonym z dwóch znaków dwukropka **::** .

W przykładzie 7 zademonstrowano implementację lokalną metody *odbierz*() (linia 4). W celu zachowania czytelności umieszczono całość w jednej linii, jednak widać że definicja ta jest identyczna z definicją funkcji według reguł języka C.

Przykład 7. Implementacja lokalna (*inline*) metody *odbierz*()

```
 1.  class Klient {
 2.     public:
 3.        void naleznosc(float kwota);
 4.        void odbierz(char danie[]) {printf("Odbieram i zjadam: %s...\n",danie); };
 5.        void zjedz(void);
 6.     private:
 7.        float pieniadze;
 8.        int glod;
 9.        void skorzystaj_z_baru(void);
10.  };
```

W przykładzie 8 przedstawiono implementację globalną tej samej metody *odbierz*(), w liniach 11-13. Tym razem definicja następuje po deklaracji metody (linia 4) oraz całej klasy (linie 1-10). W tym wypadku, aby zaznaczyć, że implementowana metoda należy do klasy Klient, a nie do innej klasy, ani że nie jest samodzielną funkcją, w definicji jej nazwa jest poprzedzona przez *Klient::*.

Przykład 8. Implementacja globalna metody *odbierz*()

```
 1.  class Klient {
 2.     public:
 3.        void naleznosc(float kwota);
 4.        void odbierz(char danie[]);
 5.        void zjedz(void);
 6.     private:
 7.        float pieniadze;
 8.        int glod;
 9.        void skorzystaj_z_baru(void);
10.  };
11.  void Klient::odbierz(char danie[]){
12.          printf("Odbieram i zjadam: %s...\n", danie);
13.  }
```

Implementacja metod podlega dokładnie tym samym zasadom i tej samej składni, co definicja każdej funkcji, to znaczy wymaga podania:

- typu wartości zwracanej przez metodę, lub słowa kluczowego *void*,
- nazwy metody,
- listy parametrów, w nawiasach okrągłych,
- ciała metody, ujętego w nawiasy klamrowe.

Każdą metodę można zaimplementować dowolnym sposobem: lokalnie lub globalnie; sposoby implementacji metod można mieszać w ramach jednej klasy. Niedozwolone jest jedynie zdefiniowanie tej samej metody więcej niż raz. Praktycznie jednak implementację *inline* stosuje się rzadziej, i co najwyżej wtedy, gdy metoda jest krótka, aby zbyt rozległym kodem nie zaciemniać deklaracji klasy.

5.5 Odwołanie do pól klasy

Od chwili utworzenia obiekt danej klasy istnieje w pamięci, i można zacząć korzystanie z jego metod, oraz z jego pól danych. Metodę wywołuje się niemal identycznie, jak funkcję; trzeba w tym celu:

- podać jej nazwę,
- podać listę parametrów w nawiasach okrągłych
- i ewentualnie przechwycić zwracaną wartość, jeżeli z deklaracji wynika, że zwraca wartości.

Wywoływanie metod należących do utworzonego obiektu, w porównaniu z wywoływaniem funkcji wymaga tylko jednego, logicznego uzupełnienia: skoro w jednym programie może być wiele instancji danej klasy, a każda z nich wyposażona jest w identyczne metody, trzeba wskazać, z którego obiektu ma być wywołana dana metoda. Każda metoda operuje na polach danych tylko swojego obiektu. Zatem – choć metody są identyczne – w zależności od obiektu i jego stanu, wynik

działania metody może być inny, i a ewentualne zmiany stanu powinny zostać odnotowane we właściwym obiekcie. Stąd przy wywołaniu metody jej nazwę poprzedza się nazwą obiektu, do którego przynależy.

Nazwę obiektu oddziela się od nazwy metody operatorem oznaczanym znakiem kropki `.`, jak w linii 20 w przykładzie 9. Jeżeli jednak nazwa obiektu nie jest określona, lecz tylko wskazuje na niego zmienna wskaźnikowa, trzeba najpierw dokonać dereferencji tej zmiennej z użyciem operatora gwiazdka `*`, otrzymując w ten sposób wskazywany obiekt, i dopiero wówczas zastosować operator kropka, aby z obiektu wyłuskać wywoływaną metodę. Prowadzi to do dość skomplikowanego zapisu (linia 21), stąd analogicznie jak w języku C stosuje się dodatkowy operator złożony ze znaków minusa i większości `->`, który pozwala w prostszy sposób zapisać dokładnie tę samą operację dereferencji i wyłuskania (linia 22).

Przykład 9. Wywołania metod i odwołania do pól danych

```
1.  class Klient {
2.      //znana deklaracja pominięta dla skrócenia przykładu
3.  };
4.
5.  void Klient::zjedz(void){     //implementacja globalna metody zjedz() z klasy Klient
6.    glod++;                     //kiedy klient myśli o jedzeniu, robi się bardziej głodny
7.    if(glod>=5) {
8.      printf("Ide do baru !\n");
9.      skorzystaj_z_baru();
10.   } else
11.     printf("Eee... Nie jestem glodny...\n");
12. };
13.
14. int tworzenie(){
15. Klient adam;
16. Klient *wskKlient;            //wskKlient to wskaźnik do obiektów klasy Klient
17. wskKlient = new(Klient);
18. Klient *adrian=new(Klient); //adrian to wskaźnik do obiektów klasy Klient
19.
20. adam.zjedz();                 // adam jest obiektem. Użyj z niego metody zjedz()
21. (*wskKlient).zjedz();         //wskKlient jest wskaźnikiem do obiektów, nie obiektem
22. adrian->zjedz();              //adrian  jest wskaźnikiem do obiektów, nie obiektem
23. }
24.
25. void Klient::naleznosc(float kwota){
26.   if(pieniadze>0) {
27.     pieniadze=pieniadze-kwota;
28.     printf("Place %.2f, zostalo mi %.2f...\n", kwota,pieniadze);
29.     getchar();
30.   } else {
31.     printf("Koniec pieniedzy (%.2f); koniec programu!",pieniadze);
32.     exit(1);          // koniec programu z braku pieniędzy
33.   }
34. }
```

Oczywiście mechanizm hermetyzacji nie pozwala skompilować kodu, który próbowałby wywołać prywatną metodę klasy: stąd z funkcji, które mają dostęp do instancji klasy *Klient*, można w dowolny z opisanych wyżej sposobów wywołać metody *naleznosc*(), *odbierz*() czy *zjedz*() - ale nie można wywołać metody *skorzystaj_z_baru*() - gdyż jest ona prywatna. Metodę *skorzystaj_z_baru*() są za to w stanie wywoływać wszystkie metody które są zdefiniowane w klasie Klient.

W ramach klasy, każda metoda ma pełen dostęp do wszystkich pól danych oraz może wywoływać wszystkie metody. Co więcej, wszystkie elementy klasy są dostępne bez stosowania dodatkowych operatorów:

- do każdego pola danych można odwoływać się jak do zmiennej w zasięgu widzialności. Co więcej, pola danych zachowują się jak zmienne klasy *static* – czyli zachowują poprzednią wartość pomiędzy kolejnymi wywołaniami metod klasy.
- każdą metodę tej samej klasy można wywoływać jak funkcję w zasięgu widoczności.

W przykładzie 9 w liniach 6-7 widnieje odwołanie do pola danych *glod*. Ponieważ jest ono w tej samej klasie co metoda *zjedz*(), która właśnie z niego korzysta – odwołanie do tego pola nie wymaga żadnych dodatkowych operatorów. Co więcej metoda może zmieniać tę wartość (linia 6), i taka zmieniona wartość zostanie zachowana w obiekcie do czasu wywołania metody, która ją znowu zmieni. Analogicznie w liniach 26-28 i 31 metoda *naleznosc*() sprawdza i zmniejsza zasoby finansowe stosownie do kosztów posiłku, stosując wprost nazwę pola danych *pieniądze*, którego wartość pozostaje zachowana pomiędzy jej wywołaniami.

W przykładzie 9 w linii 9 można znaleźć przypadek, gdy jedna metoda wywołuje inną metodę – *skorzystaj_z_baru*() – w ramach tej samej klasy. Niezależnie od tego, w jakiej sekcji została zadeklarowana wywoływana metoda (prywatnej, chronionej czy publicznej), w ramach tej samej klasy można ja wywołać jak zwykła funkcję, bez dodatkowych operatorów.

Nie poruszono dotąd kwestii odwołania się do pola danych spoza klasy. Składniowo poprawnie można takie odwołanie zapisać:

```
adam.pieniadze=100;        // prawdopodobnie błędne?
adrian->glod=1;            // prawdopodobnie błędne?
```

Jednak, mimo poprawnej składni, taki zapis jest niezgrabny znaczeniowo, i należy go unikać. Wszak oznaczałby, że stan obiektu modyfikowany jest spoza jego klasy, co jest zaprzeczeniem idei, i związanych z nimi korzyści, programowania obiektowego: że obiekt sam przechowuje i samo modyfikuje swój stan, zaś programista-użytkownik klasy może wpływać na stan obiektu tylko poprzez wywołanie jego publicznych metod. Stosowanie takiego zapisu wymaga dokładnego przemyślenie, gdyż na ogół wskazywałoby na poważne błędy w projekcie obiektu:

- programista-twórca klasy nie przewidział i nie udostępnił dodatkowej metody, która zmieni stan obiektu w pożądany sposób;
- programista-twórca klasy nie zastosował hermetyzacji, żeby ukryć stan obiektu, i udostępniać go tylko poprzez metody.

5.6 Konstruktor

Konsekwentna realizacja zasady, że obiekt powinien samodzielnie zmieniać swój stan powoduje problem: kiedy i jak zainicjalizować pola jego danych? Wszak metody – na przykład metoda *zjedz*() czy *naleznosc*() – bazują na tym, że zarówno ilość pieniędzy, jak i poziom głodu mają już jakieś wartości, i te wartości są modyfikowane. Już przy pierwszym wywołaniu te metody potrzebują zainicjalizowanych poprawnymi wartościami pól *pieniadze* i *glod*.

Pierwszym, i błędnym, pomysłem na miejsce inicjalizacji pól danych może wydawać się ich deklaracja: gdy w przykładzie 5 w liniach 7 i 8 deklarowane są pola *pieniadze* i *glod*, na pozór można by je od razu zainicjalizować następująco:

```
float pieniadze=100;
int glod=0;
```

Zapis byłby więc identyczny do inicjalizacji zmiennych globalnych. Jednak trzeba zauważyć, że definicja klasy jest tylko jedna. Zaś instancji tej klasy może być wiele. W programie ograniczeniem byłoby, gdyby każdy obiekt klasy *Klient* musiał na wstępie mieć dokładnie tyle samo gotówki i taki sam poziom poczucia głodu. Żeby stworzyć dwa poza tym identyczne obiekty, różniące się tylko którąś z tych cech, trzeba by dla każdego definiować oddzielną klasę. Zaś dla N różnych obiektów – N klas, a w każdej klasie te same metody. Byłoby to przeciwieństwem zasady ponownego wykorzystywania raz napisanego kodu (*code re-use*): na przykład w przypadku wykrycia błędu w którejś metodzie, trzeba by nanosić identyczną poprawkę tej metody w każdej klasie. Ogranicza się wówczas istotną zaleta programowania obiektowego: że jedna klasa może mieć wiele odmiennych instancji.

Skoro metody już przy pierwszym wywołaniu potrzebują zainicjalizowanych poprawnymi wartościami pól *pieniadze* i *glod* – to inicjalizacja musi odbyć się wcześniej – zanim zostanie użyta jakakolwiek z dotąd zdefiniowanych metod. Jednak z drugiej strony: przed wywołaniem którejkolwiek z metod nie da się zmienić wartości prywatnych pól danych, gdyż w programowaniu obiektowym tylko one mają do tych pól dostęp. Wyjściem z tego błędnego koła jest mechanizm *konstruktora*: dodatkowej metody w ramach każdej klasy, która domyślnie uruchamiana jest zawsze w chwili, kiedy powstaje obiekt.

Istnienie konstruktora jest obligatoryjne dla każdej klasy, dlatego jeżeli programista-twórca klasy nie zaimplementuje własnego konstruktora, wykonującego pożyteczne zadania, kompilator dodaje w procesie kompilacji uproszczony konstruktor który nie wypełnia żadnych zadań, tylko aby uczynić zadość formalnemu wymaganiu. Uruchomienie konstruktora nie wymaga żadnych dodatkowych zabiegów ze strony programisty, wystarczy tylko utworzyć obiekt. W powyższych przykładach zatem, ilekroć tworzono obiekt, uruchamiany był dla niego domyślny, pusty konstruktor dodawany przez kompilator.

Konstruktor uruchamia się na końcu procesu tworzenia obiektu, i może – zgodnie z zawartością swojej definicji – zainicjalizować wszelkie potrzebne pola danych. Konstruktor ma takie same możliwości jak każda inna metoda. Może wykorzystywać wszystkie dostępne funkcje, obiekty, czy konstrukcje językowe – ma więc pełne możliwości, jeśli chodzi o wyznaczenie wartości początkowych, którymi zostaną zainicjalizowane pola danych.

Zasady tworzenia konstruktora są następujące:

- nazwa konstruktora musi być taka sama jak nazwa klasy. Pozwala to jednoznacznie zidentyfikować metody, które są konstruktorami;
- konstruktor nie może zwracać żadnych wartości. Stąd w deklaracji konstruktora niedopuszczalne jest określanie typu zwracanej wartości i podanie jej powoduje błąd kompilacji;
- konstruktor przeważnie zadeklarowany jest w sekcji *public*. Wynika to z konsekwencji założenia, że tylko publiczne metody dają się wywołać programiście-użytkownikowi klasy, a zatem tylko wówczas będzie on mógł utworzyć obiekt, czemu towarzyszy uruchomienie konstruktora;
- konstruktor może mieć parametry.

W przykładzie 10 widać w liniach 6-8 implementację lokalną konstruktora klasy *Klient*. Konstruktor łatwo rozpoznać, gdyż nazywa się Klient – tak jak klasa, oraz nie ma zadeklarowanego typu zwracanych wartości. Przedstawiony konstruktor nie pobiera parametrów, i korzystając – jak

każda metoda klasy – z bezpośredniego dostępu do pól danych, inicjalizuje je stałymi wartościami. Stosowanie stałych w konstruktorze powoduje, że wszystkie instancje tej klasy zostaną zainicjalizowane takimi samymi wartościami.

Przykład 10. Implementacja lokalna prostego konstruktora

```
1.    class Klient {
2.       public:
3.          void naleznosc(float kwota);
4.          void odbierz(char danie[]);
5.          void zjedz(void);
6.          Klient(void) { pieniadze=99.99;      // prawie 100zl
7.                         glod=0;               // nie głodny
8.                       };
9.       private:
10.         float pieniadze;
11.         int glod;
12.         void skorzystaj_z_baru(void);
13.   };
```

W przykładzie 11 widać początkowo samą deklarację konstruktora w linii 6. Łatwo poznać, że jest to deklaracja konstruktora, gdyż ta metoda nazywa się Klient – tak, jak klasa, i nie ma zadeklarowanego typu zwracanej wartości. W liniach 13-20 widać z kolei jego globalną implementację.

Ten konstruktor pobiera wartości pól danych ze standardowego wejścia, a zatem prawdopodobnie od użytkownika. Stąd drukuje najpierw tekst zachęcający do wprowadzenia danych. Te elementy niczym się nie różnią od pobierania danych do zmiennych lokalnych, znanego z pierwszych kontaktów z językiem C. Dodatkowo przy wprowadzaniu wielkości *glod* konstruktor prowadzi podstawową kontrolę poprawności wprowadzonych danych i wymusza ich ponowne wprowadzenie, gdy wartość nie mieści się w zadanym przedziale (pętla *do-while* w liniach 16-19).

Przykład 11. Implementacja globalna prostego konstruktora

```
1.    class Klient {
2.       public:
3.          void naleznosc(float kwota);
4.          void odbierz(char danie[]);
5.          void zjedz(void);
6.          Klient(void);
7.       private:
8.          float pieniadze;
9.          int glod;
10.         void skorzystaj_z_baru(void);
11.   };
12.
13.   Klient::Klient(void) (){
14.         puts("Podaj ilosc pieniedzy");
15.         scanf("%f",&pieniadze);
```

```
16.         do {
17.               puts("Okresl glod (0 = nie glodny ; 5 = bardzo glodny");
18.               scanf("%d",&glod);
19.         while(glod<0 || glod>5);       // pytaj aż wartość znajdzie się w przedziale od 0 do 5
20.     }
```

Najbardziej jednak interesującą i najczęściej spotykaną formą konstruktora jest konstruktor wykorzystujący parametry. Konstruktor z dwoma parametrami: pierwszym typu float, i drugim typu int, jest zadeklarowany w linii 6 przykładu 12. Jego implementacja w liniach 13-16 jest bardzo prosta, bo polega jedynie na przepisaniu wartości parametrów do odpowiadających im pól danych klasy. Konstruktor – jako metoda należąca do klasy – ma prawo to zrobić, podczas gdy w funkcji *main*() byłoby to niewykonalne, ze względu na hermetyzację obiektu.

Przykład 12. Implementacja globalna konstruktora z parametrami

```
1.      class Klient {
2.         public:
3.            void naleznosc(float kwota);
4.            void odbierz(char danie[]);
5.            void zjedz(void);
6.            Klient(float p, int g);
7.         private:
8.            float pieniadze;
9.            int glod;
10.           void skorzystaj_z_baru(void);
11.     };
12.
13.     Klient::Klient(float p, int g){
14.           pieniadze=p;
15.           glod=g;
16.     }
17.
18.     int main(){
19.     Klient adam(50, 1);       // obiekt adam będzie miał początkowo 50zl i będzie nieco głodny
20.
21.     do{
22.        adam.zjedz();
23.     }while(1);          // wieczna pętla. Zupełnie jak Mama.
24.
25.     return EXIT_SUCCESS; // oznacza poprawne zakończenie programu
26.     }
```

Paradoksalnie taka nieskomplikowana implementacja pozostawia jednak największe możliwości: to programista-użytkownik klasy ma w pełni kontrolę nad tym, jako określi wartości do zainicjalizowania obiektu: może deklaracją obiektu widoczną w linii 19 utworzyć obiekt i zainicjalizować go stałymi wartościami. Może jednak również utworzyć obiekt stosując wartości zadeklarowanych wcześniej zmiennych, pochodzące od użytkownika czy uzyskane w dowolny inny sposób:

```
Klient adam(finanse, laknienie);
```

Konstruktor oprócz inicjalizacji pól danych może podczas tworzenia obiektu wykonywać wiele innych zadań, na przykład otwierać okienko graficzne na ekranie czy dynamicznie alokować pamięć. W chwili kiedy obiekt przestaje istnieć, pojawia się potrzeba, aby „posprzątać po nim", czyli cofnąć dokonane na jego rzecz zmiany: zamknąć okienko graficzne, zwolnic zaalokowaną pamięć, itp. Stąd każda klasa posiada również druga specjalną metodę, zwaną destruktorem. Destruktor uruchamiany jest zawsze domyślnie podczas usuwania obiektu, bez żadnych zabiegów ze strony programisty-użytkownika klasy. Jeżeli programista-twórca klasy nie zdefiniuje własnego destruktora, kompilator w procesie kompilacji dodaje domyślny własny destruktor, który nie pełni wówczas innych funkcji poza spełnieniem wymagań formalnych.

Zasady tworzenia destruktora są następujące:

- nazwa destruktora musi być taka sama jak nazwa klasy, poprzedzona znakiem tyldy ~ ;
- destruktor nie może zwracać żadnych wartości. Stąd w deklaracji destruktora niedopuszczalne jest określanie typu zwracanej wartości i podanie jej powoduje błąd kompilacji;
- destruktor przeważnie zadeklarowany jest w sekcji *public*. Wynika to z konsekwencji założenia, że tylko publiczne metody dają się wywołać programiście-użytkownikowi klasy, a zatem tylko wówczas będzie możliwe usuwanie obiektów, czemu towarzyszy uruchomienie destruktora;
- destruktor nie może mieć parametrów.

Na przykładzie 13 widać zastosowanie konstruktora i destruktora klasy do przygotowania, a następnie zwolnienia pamięci na zadaną ilość elementów typu char. Przy tworzeniu obiektu, pamięć zostanie samoczynnie zaalokowana poprzez konstruktor. Bez zastosowania destruktora przydzielona pamięć nie zostałaby zwolniona podczas usuwania obiektu.

Przykład 13. Zastosowanie destruktora do zwolnienia pamięci podczas likwidowania obiektu

```
1.   class Alokator {
2.     public:
3.       Alokator(int r);   // konstruktor
4.       ~Alokator(void);  // destruktor
5.       //reszta sekcji pominięta
6.     private:
7.       char *bufor;
8.       //reszta sekcji pominięta
9.   };
10.  Alokator::Alokator(int r){
11.       bufor=new char[r];   //alokacja miejsca na r elementów typu char
12.  }
13.  Alokator::~Alokator(void){
14.       delete(bufor);          //zwolnienie miejsca
15.  }
```

5.7 Dziedziczenie

W założeniach programowania obiektowego duże znaczenie miało ponowne wykorzystywanie raz napisanego kodu (*code re-use*). Stąd wprowadzono mechanizmy ułatwiające tworzenie klas, poprzez bazowanie na klasach istniejących. Najprostszą formą wykorzystania istniejących klas do

zbudowania nowej jest kompozycja (*composition*): można zadeklarować obiekt dowolnej istniejącej klasy jako pole w nowej klasie. W ten sposób jednym z pól klasy jest cały obiekt innej klasy, ze wszystkimi swoimi metodami i polami danych.

Bardziej zaawansowanym mechanizmem jest dziedziczenie. Dziedziczenie pozwala wykorzystać istniejącą klasę, nazywaną w tym wypadku klasą bazową do stworzenia nowej klasy, zwanej klasą pochodną (*derived*) lub potomną. Klasa pochodna jest bardziej wyspecjalizowana niż klasa bazowa: dziedziczy – czyli przejmuje – z bazowej jej charakterystykę i metody, które następnie dopasowuje się do potrzeb.

Najczęściej przydatnym sposobem dziedziczenia jest dziedziczenie publiczne. Aby wskazać, że owo tworzona klasa ma dziedziczyć w sposób publiczny po innej klasie, podaje się w deklaracji klasy znak dwukropka **:** , po nim słowo kluczowe *public*, a po nim nazwę klasy bazowej (przykład 14).

Przykład 14. Dziedziczenie publiczne

```
1.      class Mikrofalowka : public Toaster {
2.          //reszta deklaracji pominięta
3.      };
```

W przykładzie 14 deklarowana jest nowa klasa *Mikrofalowka*, na podstawie klasy *Toaster*. Klasa *Mikrofalówka* jest więc klasą pochodną (potomną), a klasa *Toaster* – klasą bazową, z której zostaną zapożyczone elementy. Dziedziczenie w żaden sposób nie wpływa na klasę bazową – jej elementy nie ulegają żadnej zmianie z powodu stworzenia klasy pochodnej. Za to w klasie pochodnej pojawiają się elementy zadeklarowane i zaimplementowane wcześniej w klasie bazowej, w dokładnie tych samych sekcjach (*public*, *private*, *protected*) w których się znajdowały.

Jednak możliwość dostępu do tych elementów w klasie pochodnej zmieni się, w stosunku do klasy bazowej; elementy pochodzące z klasy bazowej z sekcji:

- publicznej (*public*) będą w klasie pochodnej dostępne zarówno dla programisty-twórcy klasy, jak i dla przyszłego programisty-użytkownika klasy;
- prywatnej (*private*) będą w klasie pochodnej niedostępne dla programisty-twórcy klasy, ani dla przyszłego programisty-użytkownika klasy;
- chronionej (*protected*) będą w klasie pochodnej dostępne wyłącznie dla programisty-twórcy klasy.

Na przykładzie 15 widać, jak można utworzyć klasę *Mikrofalówka* na podstawie klasy *Toaster*. Założeniem jest, że choć urządzenia różnią się zasadą działania, to z punktu widzenia obsługi klienta oba służą do podgrzewania dania przez określony czas, którego upływ sygnalizują. Za jedyną istotną między nimi różnicę przyjęto, że czas nagrzewania przez toster nie jest regulowany, zaś czas podgrzewania przez mikrofalówkę jest regulowany. Tym samym można powiedzieć, że mikrofalówka jest bardzo podobna do toster, i jedynie wydłuża jego definicję, doprecyzowując pewien aspekt jego działania.

Przykład 15. Tworzenie klasy przez rozszerzenie (doprecyzowanie) klasy istniejącej

```
1.      #include <stdio.h>
2.      #include <windows.h>
3.      class Toaster {
4.         public:
```

```
 5.          void start(void);
 6.          Toaster(int czasG) { czasGrzania=czasG; }; /* czas grzania oddzielny dla każdej
   instancji Toastera
 7.       protected:
 8.          void czekaj(int czas) { Sleep(czas*1000); }
 9.          void sygnalizuj(void) { puts("gotowy"); }
10.          int czasGrzania;
11.    };
12.
13.    void Toaster::start(void) {
14.       czekaj(czasGrzania);
15.       sygnalizuj();
16.    }
17.
18.    class Mikrofalowka : public Toaster {
19.       public:
20.          Mikrofalowka(int czasG) : Toaster(czasG) {};
21.          void ustawCzas(int czasG) {czasGrzania=czasG; }
22.    };
```

Klasa *Toaster* w części publicznej (linie 5,6) przewiduje przyjmowane tylko jednego komunikatu: *start*(). Dodatkowo w części publicznej jest jeszcze tylko konstruktor, który pozwala zainicjalizowac czas podgrzewania, oddzielnie dla każdej tworzonej instancji toster. Można więc na przykład utworzyć dwa obiekty klasy *Toaster*:

```
Toaster tosterNaLadzie(60);
Toaster tosterWKuchni(300);
```

z których każdy będzie grzał przez odmienny czas – toster na ladzie 60 sekund, tylko do podgrzania już gotowego dania; zaś toster w kuchni – 300 sekund.

Metoda start(), zaimplementowana globalnie w liniach 13-16 polega na tym, aby odczekać zadany czas od chwili otrzymania komunikatu start, a następnie zasygnalizować koniec grzania. Wykorzystywane do tego metody *czekaj*(), *sygnalizuj*(), oraz pole danych *czasGrzania* są prywatne – nie potrzebuje do nich dostępu nikt, poza samym tosterem. Ponieważ wykorzystują one funkcję *puts*() oraz *Sleep*(), zostały w liniach 1 i 2 dołączone pliki nagłówkowe, definiujące te funkcje.

Klasa *Mikrofalówka* poszerza (linie 18-22), a zatem uszczegóławia, deklarację klasy *Toaster* tylko o dodanie jednej metody publicznej: *ustawCzas*(), służącej do ustawiania czasu grzania. Zatem w przeciwieństwie do tostera, czas grzania mikrofalówki można ustawić w każdej chwili, a nie tylko w momencie tworzenia obiektu. Czas grzania mikrofalówki można ustawić także w momencie tworzenia obiektu – służy do tego jej konstruktor (linia 20), który sam nie robi nic, a deklaracja jedynie wskazuje na konieczność użycia konstruktora klasy bazowej do ustawienia czasu.

6. Obiektowy kalkulator w odwrotnej notacji polskiej

6.1 Odwrotna notacja polska

Wyrażenia z dwuargumentowymi operacjami arytmetycznymi, jak np., dodawanie, odejmowanie,

mnożenie czy dzielenie zapisuje się klasycznie w notacji infiksowej, znanej z lekcji matematyki. Polega ona na tym, że operatory umieszcza się pomiędzy operandami, na których odbywa się działanie, na przykład:

2 + 2

W przypadku gdy konieczne jest wskazanie kolejności wykonywania operacji, trzeba stosować w niej nawiasy, na przykład:

(2 + 2) * 3 oznacza coś innego niż 2 + 2 * 3

Notacja polska nosi nazwę od osoby jej twórcy, Jana Łukasiewicza. Na jej podstawie powstała odwrotna notacja polska (Reverse Polish notation, RPN), zwana także postfixową. Polega ona na umieszczaniu operatorów za operandami, na przykład:

2 2 + oznacza: weź 2, a następnie 2, i je dodaj

Więcej przykładów podano w tabeli 1.

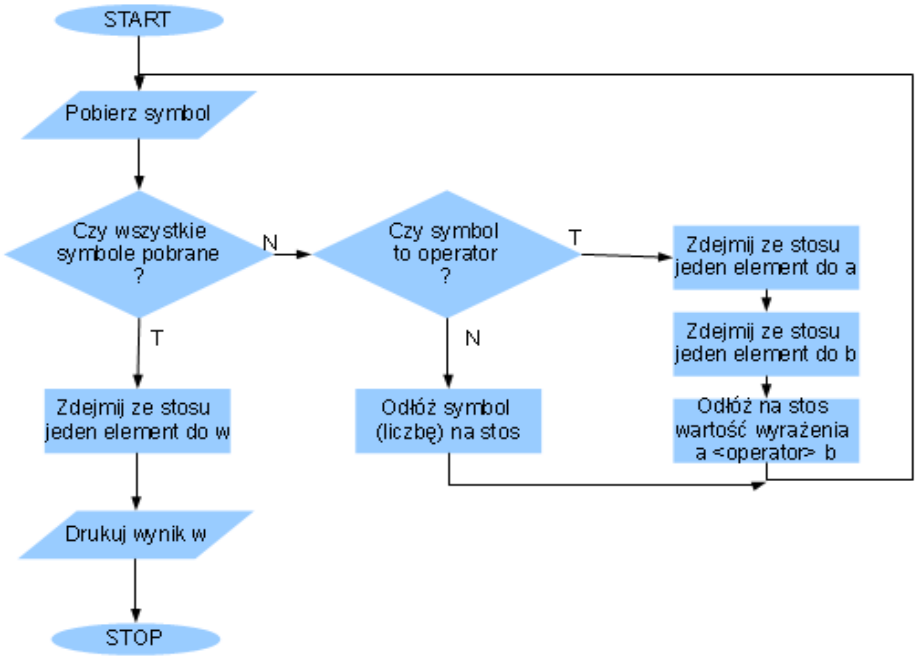
Tabela 1. Przykłady RPN

l.p.	*RPN*	*Znaczenie*	*Notacja infiksowa*
1.	***2 2 +***	***weź 2, a następnie 2, i je dodaj***	***2 + 2***
2.	***2 2 + 3 ****	***weź 2, a następnie 2, i je dodaj; weź uzyskany wynik i 3, i je pomnóż***	***(2 + 2) * 3***
3.	***2 3 * 2 +***	***weź 2, a następnie 3, i je wymnóż; weź uzyskany wynik i dodaj 2***	***2 * 3 + 2***
4.	***2 3 + 3 2 -****	***weź 2, a następnie 3, i je dodaj; weź 3, a następnie 2 i je odejmij; wymnóż wyniki***	***(2 + 3) * (3 – 2)***

Jak widać w RPN nie ma potrzeby stosowania nawiasów, co ułatwia analizowanie wyrażeń w tej notacji przez programy komputerowe, z wykorzystaniem stosu.

Algorytm obliczania wartości wyrażenia RPN przedstawiony jest na rys.5. Przez symbol rozumie się w nim dowolny ciąg znaków poprzedzony spacją lub tabulacją, zaś zakończony spacją, tabulacją lub znakiem nowej linii. W algorytmie pomija się przypadek wprowadzenia niepoprawnych danych, a zatem zakłada się, że wszystkie napotkane symbole to poprawnie przedstawione liczby lub dopuszczalne operatory dwuargumentowe (np. +, -, *, /). Proponowane rozwiązanie nie jest złożone, dzięki wykorzystaniu w nim stosu.

Algorytm polega na tym, aby pobierać kolejno wszystkie dostępne symbole. Jeżeli symbol nie jest operatorem, to wiadomo, że jest liczbą, i wartość tej liczby zostaje odłożona na stos. W przypadku poprawnych danych wejściowych oznacza to, że najwyżej dwa kolejne symbole mogą być liczbami, zaś kolejnym symbolem będzie operator. Natrafienie na operator powoduje zdjęcie ze stosu dwóch ostatnich wartości, wykonanie na nich operacji zgodnej z operatorem, i odłożenie wyniku na stosie.



Rysunek 5. Algorytm obliczania wartości wyrażenia RPN

Jeżeli teraz nastąpiłby koniec symboli (jak w wierszu 1 tabeli 1), to zostałby wydrukowany element ze stosu, czyli wynik.

Jeżeli jednak teraz nastąpiłaby liczba, jej wartość zostałaby – jako kolejna – odłożona na stos. Jeżeli po tej liczbie nastąpiłby operator (jak w wierszu 2 lub 3 tabeli 1) – dwie ostatnie wartości zostałyby pobrane ze stosu, wykonana na nich operacja, i wynik odłożony na stosie. Jeżeli zaś po tej liczbie nastąpiłaby kolejna liczba (jak w wierszu 4 tabeli 1), to zostanie ona odłożona na stos, a następujący po niej operator spowoduje wykonanie operacji na dwóch ostatnich wartościach ze stosu (3-2) i odłożenie tamże wyniku. W ten sposób na stosie znalazłyby się obok siebie dwa pośrednie wyniki: (2+3) oraz (3-2). Następujący operator spowodowałby ich pobranie ze stosu, wykonanie operacji, odłożenie ostatecznego rezultatu na stos i wydrukowanie go.

6.2 Projekt modelu obiektowego

Projektując kalkulator RPN metodą obiektową można założyć, że będzie się on składać z dwóch części:

- obiektu klasy nazwanej *RPN*, który przyjmuje komunikaty: odłóż liczbę na stos, wykonaj określoną operację na liczbach na stosie, oraz zwróć liczbę z wierzchołka stosu.
- części, która będzie pobierała kolejne symbole i rozpoznawała, czy są one liczbami czy operatorami. W przypadku rozpoznania liczby, część ta będzie wysyłać do obiektu komunikat: odłóż liczbę na stos. W przeciwnym wypadku będzie wysyłać komunikat: wykonaj zadaną operację na liczbach na wierzchołku stosu. Gdy skończy się ciąg symboli, część ta będzie wysyłała do obiektu komunikat: zwróć liczbę (wynik) z wierzchołka stosu.

Patrząc dalej, obiekt klasy *RPN* ma pewne cechy stosu (metody: odłóż na stos, pobierz ze stosu), ale i jedną własną metodę (wykonaj operację na elementach stosu) która poszerza możliwości, uzyskiwane ze stosu.

W programie warto więc zacząć od zamodelowania stosu, tworząc klasę *Stack* z implementacją

jego metod. Potem stworzyć z klasy *Stack* poprzez dziedziczenie klasę pochodną *RPN*, poszerzając klasę bazową o dodatkowe metody.

Reszta programu nie będzie obiektem, lecz wykorzysta zwykłe funkcje.

6.3 Rozwiązanie

Deklarację klasy *Stack* przedstawiono w przykładzie 16. Pojawiają się w niej elementy z przykładu 2, umieszczone we właściwych sekcjach klasy.

Przykład 16. Deklaracja klasy Stack

```
1.    #define N 100
2.    class Stack {
3.       public:
4.          double pop(void);
5.          void push(double v);
6.          Stack(void) { sp=0; };
7.       private:
8.          double stack[N];
9.          int sp;
10.   };
```

Wśród metod publicznych znalazły się oczywiście *pop*() i *push*(), służące odpowiednio do odkładania wartości na szczyt stosu i pobierania wartości ze szczytu stosu. Metody te będą służyły do używania stosu, muszą więc być publicznie dostępne. Za to szczegóły realizacji stosu stały się prywatną informacją tej klasy: nie ma potrzeby, żeby ktokolwiek wnikał, jak nazywają się zmienne, w których przechowywane są wartości, ani jakie są szczegóły symulacji stosu – na przykład to, że przyjęto maksymalną wielkość stosu N.

Wskaźnik stosu *sp* wymaga jednak inicjalizacji: wszak po utworzeniu instancji klasy *Stack* stos jest pusty, i pierwsza zapisywana wartość powinna trafić na początek tablicy *stack[]*, a nie w dowolne jej miejsce (por. rys.4). Stąd prywatne pole danych sp jest inicjalizowane konstruktorem *Stack*(), zaimplementowanym lokalnie. Konstruktor daje pewność, że ilekroć zostanie utworzony nowy obiekt klasy *Stack*, wartość *sp* będzie zero.

Pozostałe metody klasy również wymagają implementacji. Zastosowano dla nich implementację globalną, uwidocznioną w przykładzie 17. Metoda *push*() wpisuje wartość przekazaną w jej argumencie typu *double* do tablicy *stack[]* w miejscu wskazywanym przez *sp*. Ponieważ tym samym miejsce to zostaje zajęte, wartość *sp* zostaje zwiększona, aby wskazywała na kolejne wolne miejsce.

Z kolei metoda *pop*() najpierw zmniejsza wartość *sp* o jeden, aby pokazywała ostatni wpisany element, nie zaś pierwsze wolne miejsce tuż za nim, a potem zwraca ów element. W efekcie – choć nie wymazano jeszcze wartości ze stosu – jej miejsce będzie traktowane jako wolne przy najbliższym wywołaniu *push*().

Przykład 17. Implementacje metod klasy Stack

```
1.      void Stack::push(double v) {
2.      stack[sp]=v;
3.      sp++;
4.      }
5.
6.      double Stack::pop(void) {
7.      sp--;
8.      return stack[sp];
9.      }
```

Metody klasy *Stack* mają dostęp do wskaźnika stosu *sp* oraz do tablicy *stack[]*, gdyż wszystkie te elementy zawarte są w tej samej klasie. Metody zostały zaimplementowane w bardzo podstawowy sposób – nie prowadzi się na przykład kontroli przepełnienia stosu (*overflow*), ani próby pobrania wartości ze stosu pustego (*underflow*).

Działanie stosu można przetestować (przykład 18), przeplatając komunikaty odkładania i pobierania wartości ze stosu, aby przekonać się, że rzeczywiście w pierwszej kolejności zawsze uzyskuje się dostęp do wartości, które jako ostatnie zostały odłożone.

Przykład 18. Testowanie działania stosu

```
1.      void test(void){
2.      class Stack stack;
3.
4.      stack.push(1.23);
5.      stack.push(2.34);
6.      printf("Pobrano: %f\n", stack.pop());
7.      stack.push(3.24);
8.      printf("Pobrano: %f\n", stack.pop());
9.      printf("Pobrano: %f\n", stack.pop());
10.     }
```

Bazując na klasie *Stack* poprzez dziedziczenie publiczne tworzy się klasę *RPN* (przykład 19). Klasa ta posiada tym samym wszystkie metody klasy *Stack*, zawarte w tych samych sekcjach. Dodatkowo w sekcji publicznej pojawiają się zaimplementowane lokalnie metody *add*(), *sub*(), *mul*(), *div*() - wszystkie skonstruowane analogicznie: dwukrotnie pobierają wartość ze stosu metodą *pop()*, wykonują na nich operację stosowną do nazwy metody, i odkładają uzyskany wynik na stos metodą *push*(). Metody *push*() i *pop*() zostały odziedziczone po klasie *Stack*, a zatem znajdują się również w klasie *RPN*.

Metoda *div*() nie kontroluje, czy wartość dzielnika jest niezerowa, co będzie prowadzić do niepożądanych wyników w przypadku pojawienia się takiej wartości.

Dodatkowo w klasie *RPN* pojawia się metoda *putInStack*(), która napis *input* uzyskany jako parametr zamienia z użyciem standardowej funkcji *strtod*() języka C (zadeklarowaną w pliku nagłówkowym stdlib.h) na liczbę typu double, którą następnie odkłada na stos wywołując *push*().

Metoda *putInStack*() będzie służyła do zapisywania na stos liczb rozpoznanych pośród symboli. Ponieważ jednak symbole będą wpisywane jako ciągi znaków, czyli łańcuchy, to na przykład liczba

100 będzie wprowadzona jako łańcuch „100” i będzie wymagała takiej zamiany.

Przykład 19. Deklaracja klasy RPN dziedziczącej publicznie po Stack

```
11.    #include <stdlib.h>
12.    class RPN:public Stack {
13.       public:
14.          void add(void) {push (pop() + pop());}
15.          void sub(void) {push (pop() - pop());}
16.          void mul(void) {push (pop() * pop());}
17.          void div(void) {push (pop() / pop());}
18.          void putInStack(char *input){ push(strtod(input, NULL));}
19.    };
```

Tym samym zadeklarowano i zaimplementowano wszystkie metody klas. Pozostaje utworzyć obiekt tej klasy, i wysyłać do niego komunikaty. Pozostałe części programu przedstawiono w przykładzie 20. Funkcja *getSymbol*() (linie 5-19) nie zawiera elementów programowania obiektowego, i napisana jest bardzo zgrubnie. Jej dogłębne zrozumienie nie jest tu istotne, choć oczywiście w pełni możliwe na podstawie dotąd poznanego materiału z języka C. Zadaniem tej funkcji jest oddzielenie poszczególnych symboli w łańcuchu znaków wpisywanym przez użytkownika. Warto jedynie zwrócić uwagę, że zwraca ona w linii 19 wartość NUMBER, zdefiniowaną w linii 3, która oznacza iż rozpoznała ciąg znaków opisujący liczbę, i umieściła go w swoim argumencie *input[]*. W przeciwnym razie zwraca kod ASCII znaku, opisującego rozpoznany operator lub znak końca linii ('\n').

Przykład 20. Pozostałe części programu

```
1.     #include <stdio.h>
2.     #include <ctype.h>
3.     #define NUMBER 0
4.
5.     int getSymbol(char input[]) {
6.     int i,c;
7.     while((input[0]=c=getchar())==' '|| c=='\t');
8.     input[1]='\0';
9.     if (!isdigit(c) && c!='.')
10.       return c;
11.    i=0;
12.    if (isdigit(c))
13.       while (isdigit(input[++i]=c=getchar()));
14.    if (c=='.')
15.       while (isdigit(input[++i]=c=getchar()));
16.    input[i]='\0';
17.    ungetc(c, stdin);
18.    return NUMBER;
19.    }
20.
```

```
21.     int main(){
22.     int input_type;
23.     char input[100];
24.     class RPN stack;
25.
26.     while( (input_type = getSymbol(input)) != '\n') {
27.       switch(input_type) {
28.         case NUMBER :
29.           stack.putInStack(input);
30.           break;
31.         case '+':
32.           stack.add();
33.           break;
34.         case '-':
35.           stack.sub();
36.           break;
37.         case '*':
38.           stack.mul();
39.           break;
40.         case '/':
41.           stack.div();
42.           break;
43.       } //koniec switch()
44.     }     // koniec while()
45.     printf("\nWynik=%f\n",stack.pop());
46.     getchar();
47.     }     // koniec main()
```

W funkcji *main*() w linii 24 tworzony jest obiekt *stack* klasy *RPN* o lokalnym zasięgu. W tym momencie uruchamia się konstruktor tej metody, odziedziczony po klasie *Stack*, który inicjalizuje wartość wskaźnika stosu, czyli oznacza stos jako na razie pusty.

Funkcja *main*() składa się w zasadzie z jednej pętli *while*(), która działa dopóki łańcuch wejściowy nie zostanie przeanalizowany aż do znaku końca linii ('\n'). Wyniki analizy łańcucha wejściowego, zwracane przez *getSymbol*(), umieszczane są w zmiennej *input_type* (linia 26) i na jej podstawie instrukcja *switch*() (linie 27-43) wywołuje odpowiednią metodę klasy *RPN*:

- gdy rozpoznana jest liczba, wówczas łańcuch znaków opisujący tę liczbę, zawarty w tablicy znakowej *input[]* zostaje przekazany do metody *putInStack*(), w celu zamiany na wartość liczbową i odłożenia na stos;
- w pozostałych przypadkach wywoływane są metody służące do dodawania, odejmowania, mnożenia lub dzielenia wartości znajdujących się na stosie, stosownie do wykrytego operatora – jednego z **+ , - , * , /** .

Po zakończeniu pętli ostateczny wynik znajduje się na stosie. Dlatego w linii 45 jest stamtąd pobierany metodą *pop*() z klasy *RPN*, i drukowany. Funkcja *getchar*() w linii 46 służy do zatrzymania programu, aby zapobiec zamykaniu jego okna w systemie operacyjnym Microsoft Windows.

Referencje

[1] Brian W. Kernighan, Dennis M. Ritchie, *Język ANSI C*, ISBN 83-204-2979-X, Wydawnictwa Naukowo-Techniczne 2004.
[2] Brian W. Kernighan, *The Practice of Programming*, ISBN 83-204-2732-0, Wydawnictwa Naukowo-Techniczne
[3] Bruce Eckel , Thinking in C++. Edycja polska, ISBN: 83-7197-709-3 , Helion 2009.
[4] Kleanthis Thramboulidis, A Constructivism-Based Approach to Teach Object-Oriented Programming, Journal of Informatics Education and Research, Volume 5, No 1, Spring 2003.
[5] Jerzy Grębosz, Symfonia C++. Programowanie w języku C++ orientowane obiektowo. t 1,2,3
[6] Nicolai Josuttis, C++. Programowanie zorientowane obiektowo. Vademecum profesjonalisty, 83-7361-195-9, Helion 2003.
[7] Kayshav Dattatri, Język C++. Efektywne programowanie obiektowe, Helion, 2005.

Materiały graficzne

[1] Zdjęcie pobrane z http://www.catalogs.com/info/bestof/top-10-facts-about-fast-food-and-culture (2011.11.01)
[2] Zdjęcie pobrane z http://www.visitgdansk-oliwa.com/index.php/Gastronomia_w_gdansku _ oliwie (2011.11.01)
[3] Zdjęcie pobrane z http://www.discoveringukraine.com/little-known-facts-about-ukraine (2011.11.01)