6
Funkcje

Często mamy do czynienia z sytuacją, w której pewien fragment kodu musi być wykonany w różnych miejscach programu. Należy taki fragment nazwać i wydzielić jako funkcja. Dzięki temu w miejscach, gdzie miał być wykonany wystarczy umieścić instrukcje wywołania funkcji.

DRAFT

Działanie takiego wyróżnionego fragmentu kodu może zależeć od wartości parametrów (jednego lub więcej). Dlatego funkcja może mieć zadeklarowane parametry od wartości których, zależy jej działanie.

Funkcja nie tylko wykonuje pewien kod ale również może zwracać wynik. W takiej sytuacji w jej deklaracji należy podać typ wyniku.

6.1 Deklaracja i definicja funkcji

Deklaracja funkcji ma w języku C następującą postać:

```
T nazwa(T1 p1; T2 p2; ...; Tn pn);
```

gdzie `T` jest typem zwracanego wyniku, `nazwa` jest nazwą deklarowanej funkcji, `Ti pi` jest deklaracją *i*-tego parametru o nazwie `pi` i typie wartości `Ti`.

Sama deklaracja funkcji informuje jedynie kompilator, że w programie pojawi się taka funkcja. Do skorzystanie z niej konieczna jest jeszcze definicja.

Definicja funkcji ma postać:

```
T nazwa(T1 p1; T2 p2; ...; Tn pn)
{
 Blok;
}
```

gdzie `Blok` jest blokiem instrukcji stanowiących treść funkcji. Instrukcje te wykonywane są gdy funkcja zostanie wywołana.

Blok taki powinien zawierać instrukcje `return Wyr` (jedną lub więcej), które kończą wykonywanie treści funkcji a wartość wyrażenia jest zwracanym wynikiem funkcji (powinna ona być takiego typu `T` jak podano w definicji).

Jeśli funkcja ma wynik typu `T` ale jej działanie nie kończy się odpowiednią instrukcją `return Wyr`, to taką definicję funkcji należy uznać za błędną (nieokreślony wynik funkcji).

6.2 Procedura

Szczególnym rodzajem funkcji są takie, które nie zwracają wyniku. Odpowiadają one procedurom z innych języków programowania (np. Pascal albo Ada).

Funkcję taką deklaruje się podając jako typ wyniku pusty typ `void`. Typ ten nie posiada żadnych wartości.

6.3 Wywołanie funkcji

Wywołanie funkcji ma w języku C postać:

```
nazwa(w1, w2, ..., wn)
```

gdzie `wi`, dla $i = 1, 2, \ldots, n$, jest wyrażeniem, którego wartość zostanie przekazana jako wartość i-tego parametru funkcji (powinna ona być odpowiedniego typu `Ti` jaki podano w jej deklaracji).

W szczególnym przypadku, gdy funkcja nie ma parametrów, to jej wywołanie ma postać:

```
nazwa()
```

DRAFT

Konieczne jest podanie pustych nawiasów `()` (inaczej niż w innych językach np. Pascal albo Ada).

Jeśli funkcja zwraca wynik, to najczęściej występuje w kontekście (miejscu) wyrażenia i zwrócona przez nią wartość zostanie użyta w wyliczanym wyrażeniu.

W przypadku funkcji o pustym typie wartości `void` musi ona wystąpić w kontekście instrukcji.

Często w programach korzystać będziemy z samych definicji funkcji bez podawania ich deklaracji. Jednak deklaracje są niezbędne w przypadku plików nagłówkowych (zapowiadają one jakie funkcje są dostępne w dołączanych bibliotekach) oraz w przypadku funkcji wzajemnie wywołujących się.

Rozpatrzmy następujący przykład, w którym funkcja `f()` wywołuje funkcję `g()` a funkcja `g()` wywołuje funkcję `f()`:

```
void f()
{
  ...
  g();
  ...
}

void g()
{
  ...
  f();
  ...
}
```

Aby kompilator mógł bez problemu skompilować te dwie funkcje, to przed kompilacją funkcji `g()` musi znać deklarację funkcji `f()`. Jednak podobnie przed kompilacją funkcji `f()` musi znać on deklarację funkcji `g()`. Aby przerwać ten cykl wzajemnych zależności wystarczy wcześniej podać deklaracje funkcji (przed ich definicjami):

```
void f();
void g();

void f()
{
  ...
  g();
  ...
}

void g()
{
  ...
  f();
  ...
}
```

DRAFT

6.4 Przekazywanie parametrów

W różnych językach programowania można spotkać różne tryby przekazywania wartości parametrów. Na przykład w języku Pascal była dwa tryby: przez wartość i przez zmienną; natomiast w języku Ada są trzy tryby: `in`, `out` i `in out`[1].

Język C jest pod tym względem ubogi i ma tylko jeden tryb przekazywania parametru: **przez wartość**. Oznacza to, że jedyne co możemy, to przekazać funkcji jaką wartość ma przyjąć dany parametr.

Zdarza się jednak, że musimy użyć parametrów do odebrania wartości z funkcji.

Na przykład wywołując funkcję `scanf()` oczekujemy, że ta przeczyta wpisaną przez nas liczbę i umieści ją w zadanej zmiennej.

Aby funkcja mogła oddać na zewnątrz (do miejsca wywołania) parametrem wartość, konieczne jest zadeklarowanie typu tego parametru jako wskaźnikowego. W ten sposób funkcja wie gdzie odłożyć wartość. Nadal parametr taki jest przekazywany przez wartość bo jego wartością jest dostarczane mu wskazanie.

Objaśnimy to dokładnie krok po kroku na przykładzie następującego programu:

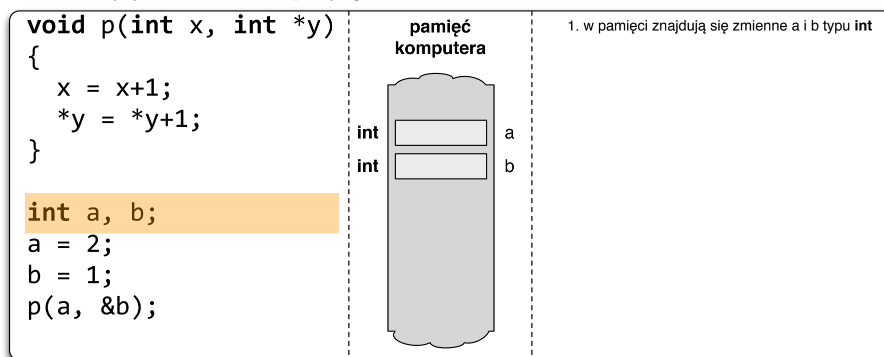
```
void p(int x, int* y)
{
  x = x + 1;
  *y = *y + 1;
}
...
```

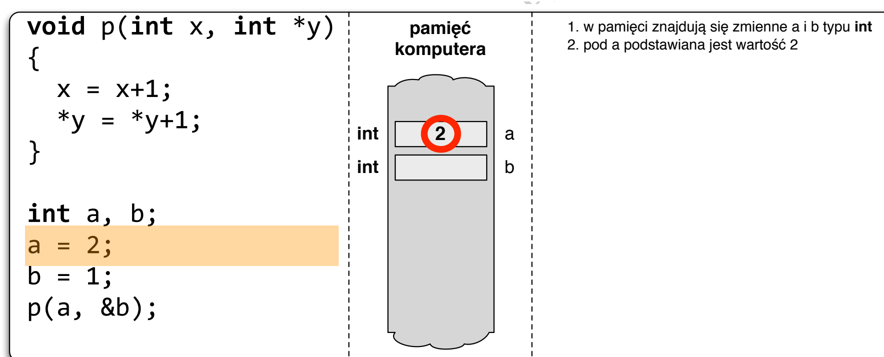
[1] Tryb w którym dostarczamy wartość, tryb w którym odbieramy wartość i tryb w którym zmieniamy wartość.

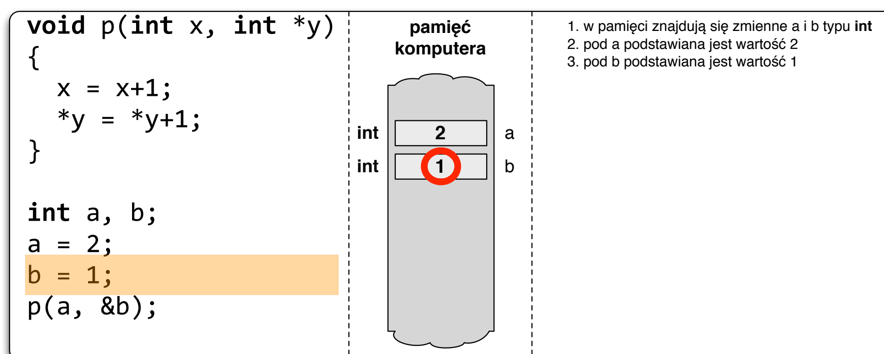
```
    int a, b;
    a = 2;
    b = 1;
    p(a, &b);
...
```

Powyższy program wyjaśnia różnicę między przekazaniem wartości całkowitej a przekazaniem wskazania do wartości całkowitej.

Na kolejnych rysunkach przedstawiono co dzieje się przed wywołanie funkcji, w trakcie jej obliczania i po jego zakończeniu.







```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int	2	a
int	1	b

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int	2	a
int	1	b
int		x

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int	2	a
int	1	b
int		x
int*		y

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int	2	a
int	1	b
int	2	x
int*		y

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int	2	a
int	1	b
int	2	x
int*		y

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int	2	a
int	1	b
int	2	x
int*		y

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int	2	a
int	1	b
int	3	x
int*		y

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int	2	a
int	2	b
int	3	x
int*		y

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```


1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

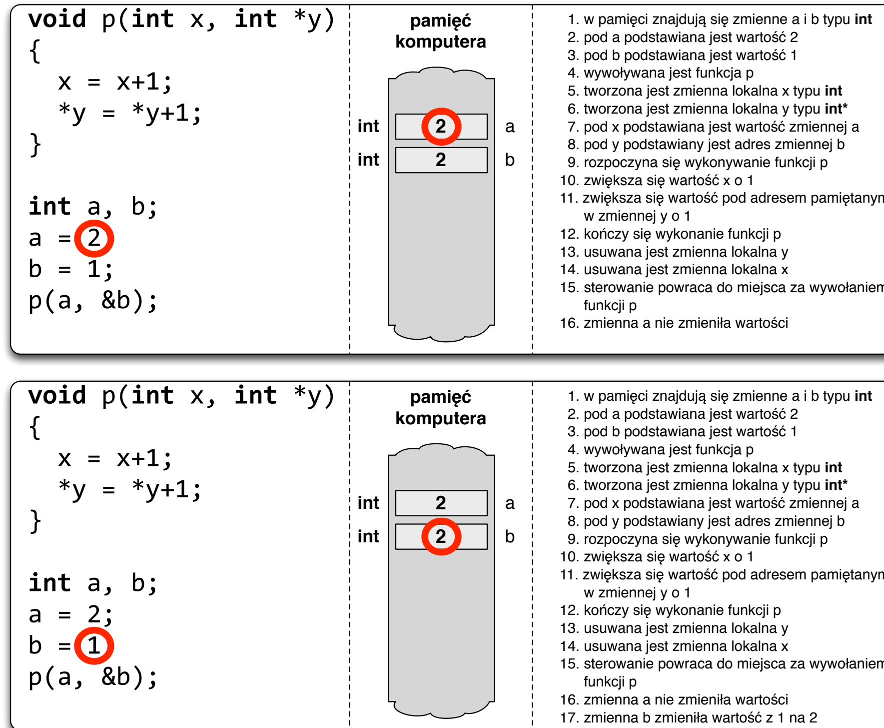

1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```


1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x

```
void p(int x, int *y)
{
  x = x+1;
  *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```


1. w pamięci znajdują się zmienne a i b typu **int**
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu **int**
6. tworzona jest zmienna lokalna y typu **int***
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x
15. sterowanie powraca do miejsca za wywołaniem funkcji p



6.4.1 Przykłady

Poniższa funkcja `swap()` dokonuje wymiany dwóch wartości całkowitych:

```
void swap(int* x, int* y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

Przykładowe wywołania funkcji `swap()`:

```
int a, b;
int t[10];
...
swap(&x, &y);
swap(&t[i], &t[j]);
swap(&x, &t[k]);
```

Poniższa funkcja `safe_sqrt()` liczy pierwiastek kwadratowy z danej wartości. Wartością funkcji jest status wyniku (czy jest poprawnie policzony).

```
bool safe_sqrt(double x, double* result)
{
  if(x >= 0.0)
  {
    *result = sqrt(x);
    return true;
  }
```

```
  return false;
}
```

Przykład użycia funkcji `safe_sqrt()`:

```
double delta = b*b - 4*a*c;
double sqrt_delta;
if(safe_sqrt(delta, &sqrt_delta))
{
  double x1 = (-b + sqrt_delta)/(2*a);
  double x2 = (-b - sqrt_delta)/(2*a);
  // tutaj obsługa pierwiastków x1 i x2
}
else
 // tutaj nie można korzystać z wartości sqrt_delta
```

6.5 Przekazywanie tablic

Rozpatrzmy następujący przykład funkcji obliczającej sumę wartości w danej tablicy liczb całkowitych:

```
int suma1(int n, int* ptr)
{
  int acc = 0;
  for(int i = 0; i < n; i++)
    acc = acc + ptr[i];       // ptr[i] == *(ptr + i)
  return acc;
}
```

DRAFT

Przykład wywołania powyższej funkcji:

```
int t[5] = {1, 3, 2, 5, 4};
int x = suma1(5, t);
```

Do funkcji `suma1()` zostaje przekazana liczba elementów tablicy (parametr `n`) i wskazanie na początek tablicy (parametr `ptr`). Sama nazwa tablicy jest wskazaniem na jej początek więc nie należy w wywołaniu dodawać operator `&`.

Powyższą funkcję można zapisać trochę inaczej:

```
int suma2(int n, int tab[])
{
  int acc = 0;
  for(int i = 0; i < n; i++)
    acc = acc + tab[i];
  return acc;
}
```

Powyższa funkcją jest funkcjonalnie identyczna z poprzednią ale w teg drugiej już sam nagłówek funkcji sugeruje, że drugi parametr służy do przekazania tablicy wartości całkowitych a nie do odebrania pojedynczej wartości całkowitej.

Funkcję `suma2()` wywołuje się dokładnie tak samo jak funkcję `suma1()`:

```
int t[5] = {1, 3, 2, 5, 4};
int x = suma2(5, t);
```

W obu funkcjach `suma1()` i `suma2()` drugi parametr jest po prostu wskazaniem na liczbę całkowitą.

Od roku 1999 jest możliwe zadeklarowanie parametru funkcji jako tablicy zmiennej długości.

C99

Trzecia wersja funkcji sumującej elementy tablicy korzysta z tej możliwości:

```
int suma3(int n, int tab[n])
{
  int acc = 0;
  for(int i = 0; i < n; i++)
    acc = acc + tab[i];
  return acc;
}
```

Tym razem kompilator tłumacząc funkcję `suma3()` wie, że drugi parametr jest wskazaniem na początek tablicy n-elementowej a nie jedynie wskazaniem na liczbę całkowitą. Zwiększa to również czytelność kodu. Osoba czytająca tekst źródłowy od razu wie, że drugim parametrem funkcji jest tablica i ma ona n elementów.

Wywołanie funkcji `suma3()` jest dokładnie takie samo jak dwóch poprzednich:

```
int t[5] = {1, 3, 2, 5, 4};
int x = suma3(5, t);
```

DRAFT

6.6 Przykłady

Na zakończenie zaprezentujemy kilka przykładów użycia funkcji.

6.6.1 Algorytm Euklidesa

Na listingu 6.1 przedstawiono funkcję obliczającą największy wspólny dzielnik. Funkcja `nwd()` oczekuje dwóch dodatnich argumentów całkowitych i zwraca wartość całkowitą. Przed rozpoczęciem obliczeń funkcja `nwd()` upewnia się, że oba jej argumenty są liczbami dodatnimi.

Listing 6.1: Algorytm Euklidesa

```
// euklide.c
//
// Obliczenie największego wspólnego dzielnika algorytmem
// Euklidesa.

#include <stdio.h>
#include <assert.h>

int nwd(int a, int b)
{
  while(b > 0)
  {
    // niezmiennik: nwd(m, n) = nwd(a, b)
    int reszta = a % b;
    a = b;
    b = reszta;
  }
```

```
  return a;
}

int main(void)
{
  int m, n;
  printf("Podaj pierwszą dodatnią liczbę całkowitą: ");
  scanf("%d", &m);
  assert(m > 0);
  printf("   Podaj drugą dodatnią liczbę całkowitą: ");
  scanf("%d", &n);
  assert(n > 0);
  printf("nwd(%d, %d) = %d\n", m, n, nwd(m, n));
  return 0;
}
```

6.6.2 Sortowanie przez wstawianie

Na listingu 6.2 przedstawiono funkcję `sort()`, która sortuje tablicę wartości całkowitych algorytmem sortowania przez wstawianie. Algorytm ten w pętli zewnętrznej wstawia kolejne elementy tablicy `tab[i]`, dla $i = 1, 2, \ldots, n-1$, do uporządkowanego fragmentu tablicy `tab[0..i]`. Niezmiennikiem tej pętli jest warunek:

$$\forall_{j=0}^{i-1}\ tab[j] \leq tab[j+1].$$

DRAFT

Listing 6.2: Sortowanie przez wstawianie

```
// wstawsort.c
//
// Algorytm sortowania przez wstawianie.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void sort(int dl, int tab[dl])
{
  for(int i = 1; i < dl; i++)
  {
    // wstawienie tab[i] do uporządkowanego
    // ciągu tab[0], ..., tab[i-1]
    int j = 0;
    while(j < i && tab[i] > tab[j])
      j++;
    if(j < i)
    {
      int tmp = tab[i];
      for(int k = i - 1; k >= j; k--)
        tab[k+1] = tab[k];
      tab[j] = tmp;
    }
  }
}

int main(void)
```

```
{
  int n = 20;
  int t[n];
  srand(time(NULL));
  for(int i = 0; i < n; i++)
    t[i] = rand() % 100;
  printf("przed sortowaniem: ");
  for(int i = 0; i < n; i++)
    printf("%2d ", t[i]);
  printf("\n");
  sort(n, t);
  printf("    po sortowaniu: ");
  for(int i = 0; i < n; i++)
    printf("%2d ", t[i]);
  printf("\n");
  return 0;
}
```

Przed wywołaniem funkcji program wypełnia w funkcji `main()` tablicę `t[20]` pseudolosowymi wartościami całkowitymi. Dzięki funkcją `srand()` i `time()` generowane liczby są za każdym razem inne, jeśli między kolejnymi uruchomieniami upłynęła więcej niż jedna sekunda. Funkcja `time()` dostarcza liczbę sekund jaka upłynęła od godziny 0:00 w dniu 1 stycznia 1970 roku, natomiast funkcja `srand()` ustala ziarno generatora liczb pseudolosowych[2].

DRAFT

6.6.3 Wyszukiwanie liniowe

Na listingu 6.3 przedstawiono funkcję `szukaj()`, która wyszukuje w danej tablicy zadaną wartość. Jeśli wartość zostanie znaleziona, to zwracana jest wartość `true` a czwartym parametrem można odebrać pozycję jej wystąpienia. Funkcja ta w najgorszym przypadku musi przejrzeć całą tablicę. Jej czas działania w takim przypadku jest funkcją liniową rozmiaru tablicy n i stąd taka nazwa użytego algorytmu.

Listing 6.3: Wyszukiwanie liniowe

```
// wyszukiniowe.c
//
// Wyszukiwanie zadanej wartości w nieuporządkowanej tablicy.

#include <stdio.h>
#include <stdbool.h>

bool szukaj(int n, int tablica[n], int wart, int *indeks)
{
  *indeks = 0;
  while(*indeks < n && tablica[*indeks] != wart)
    (*indeks)++;
  return *indeks < n;
}
```

[2]Generator generuje kolejną liczbę pseudolosową na podstawie wartości tzw. ziarna generatora, która to wartość zmienia się w sposób deterministyczny z jednego wywołania na drugie. Jeśli ziarno będzie inicjowane tą samą wartością, to uzyskiwane ciągi liczb pseudolosowych będą zawsze takie same.

```c
int main(void)
{
  int i;
  int t[8] = {6, 1, 3, 8, 4, 1, 9, 3};
  if(szukaj(8, t, 9, &i))
    printf("Liczba 9 występuje w tablicy na pozycji %d.\n", i);
  else
    printf("Liczby 9 nie ma w tablicy.\n");
  if(szukaj(8, t, 2, &i))
    printf("Liczba 2 występuje w tablicy na pozycji %d.\n", i);
  else
    printf("Liczby 2 nie ma w tablicy.\n");
  return 0;
}
```

6.6.4 Wyszukiwanie binarne

Na listingu 6.4 przedstawiono funkcję `szukaj()`, która wyszukuje w danej tablicy zadaną wartość. Tym razem dla poprawnego działania tej funkcji wymagane jest aby wartości w danej tablicy były uporządkowane od najmniejszej do największej. Przy każdej iteracji pętli **while** rozmiar przeszukiwanego obszaru tablicy `tablica[lewy..prawy]` zmniejsza się o połowę. Z tego powodu algorytm takiego wyszukiwania nazywa się binarnym. W najgorszym przypadku wymaga $\lceil \log_2 n \rceil$ iteracji.

DRAFT

Listing 6.4: Wyszukiwanie binarne

```c
// wyszukbinarne.c
//
// Wyszukiwanie zadanej wartości w tablicy uporządkowanej.

#include <stdio.h>
#include <stdbool.h>

bool szukaj(int n, int tablica[n], int wart, int *indeks)
{
  // zakładamy, że elementy tablicy są posortowane niemalejąco
  int lewy = 0;
  int prawy = n - 1;
  while(lewy <= prawy)
  {
    int srodek = lewy + (prawy - lewy) / 2;
    if(wart == tablica[srodek])
    {
      *indeks = srodek;
      return true;
    }
    else
      if(wart < tablica[srodek])
        prawy = srodek - 1;
      else
        lewy = srodek + 1;
  }
  return false;
```

```
}

int main(void)
{
  int i;
  int t[8] = {1, 3, 5, 6, 6, 8, 8, 9};
  if(szukaj(8, t, 3, &i))
    printf("Liczba 3 występuje w tablicy na pozycji %d.\n", i);
  else
    printf("Liczby 3 nie ma w tablicy.\n");
  if(szukaj(8, t, 7, &i))
    printf("Liczba 7 występuje w tablicy na pozycji %d.\n", i);
  else
    printf("Liczby 7 nie ma w tablicy.\n");
  return 0;
}
```

DRAFT