# Kompleksowe Notatki z Języka C

## 1. Podstawy deklaracji i typy danych

W języku C można zadeklarować zmienne bez przypisywania im wartości. Przykład:

```
int x;
float y;
```

### Typy danych

| Typ | Rozmiar | Opis | Przykład |
|---|---|---|---|
| `int` | 2 albo 4 bajty | Liczby całkowite | `1` |
| `float` | 4 bajty | Liczby zmiennoprzecinkowe (6-7 cyfr) | `21.37` |
| `double` | 8 bajtów | Liczby zmiennoprzecinkowe (15 cyfr) | `21.37` |
| `char` | 1 bajt | Znak | `'A'` |

### Specyfikatory formatu

| Specyfikator | Typ |
|---|---|
| `%d`, `%i` | `int` |
| `%f`, `%F` | `float` |
| `%lf` | `double` |
| `%c` | `char` |
| `%s` | String (tablica znaków `char[]`) |

```
float f = 35e3; // Reprezentacja liczby 35000.000000
printf("%.1f", f); // Wyświetla 35000.0
```

**Zaokrąglanie liczb:** - `%.0f` zaokrągla do pełnych wartości (>=0.5 w górę, <0.5 w dół).

### Rzutowanie

```
float a = (float)6; // 6.000000
```

### Zmienne statyczne (`static`):

- Zachowują swoją wartość pomiędzy wywołaniami funkcji.
- Deklaracja: `static int x;`.
- Zasięg ograniczony do pliku, jeśli zadeklarowane globalnie.

### Specyfikatory typu

Specyfikatory takie jak `unsigned` mogą być stosowane do modyfikacji typów:

```
unsigned int x = 10;  // Liczba bez znaku
long double y = 5.5;  // Liczba zmiennoprzecinkowa o podwójnej precyzji
```

**Modyfikatory:** - `unsigned` - liczby bez znaku. - `long` - rozszerzenie zakresu dla typów całkowitych i zmiennoprzecinkowych.

## 2. Operatory i sterowanie przepływem programu

### Operatory

| Operator | Nazwa | Opis | Przykład |
|---|---|---|---|
| `+` | Dodawanie | Dodaje | `x + y` |
| `-` | Odejmowanie | Odejmuje | `x - y` |
| `*` | Mnożenie | Mnoży | `x * y` |
| `/` | Dzielenie | Dzieli | `x / y` |
| `%` | Modulo | Reszta z dzielenia | `x % y` |
| `++` | Inkrementacja | Zwiększa o 1 | `x++` |
| `--` | Dekrementacja | Zmniejsza o 1 | `x--` |

**Operator `sizeof`** Zwraca ilość bajtów zajmowanych przez zmienną.

```
int x;
printf("%lu", sizeof(x)); // np. 4
```

### Operator przypisania

Operator `=` zwraca przypisywaną wartość, np.:

```
if (a = 1) {
    // Wartość a wynosi teraz 1, a warunek jest prawdziwy
}
```

### Operator warunkowy

```
int x = (a > b) ? a : b; // Zwraca a jeśli a > b, w przeciwnym razie b
```

### Instrukcje warunkowe i pętle

### Instrukcja `if`

```
if (x > 0) {
    printf("x jest dodatni\n");
} else {
    printf("x jest niedodatni\n");
}
```

### Pętla `while`

```
int i = 4;
while (i > 0) {
    printf("%d\n", i);
    i--;
}
```

### Pętla `for`

```
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

**`continue` i `break`** `continue` Pomija daną iterację a `break` wychodzi z pętli

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) continue; // Pomija parzyste liczby
    printf("%d\n", i);
    if(i == 8) break; // wychodzi z pętli jeśli i równe 8
}
```

### Switch

```
switch (x) {
    case 1:
        printf("Jeden\n");
        break;
    default:
        printf("Inna wartość\n");
}
```

**Ograniczenia:** W `switch` można używać tylko typów całkowitych (`int`, `char` itp.).

---

## 3. Tablice i wskaźniki

### Tablice

Tablice są podstawowymi strukturami danych w języku C:

```
int tab[5] = {1, 2, 3, 4, 5};
```

Tablice mogą być wielowymiarowe:

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

**Ważne:** Na końcu każdego stringa (tablicy `char[]`) jest znak `\0` (null terminator). ### Wskaźniki

Wskaźniki przechowują adresy pamięci zmiennych.

```
int x = 10;
int *ptr = &x;
printf("%d", *ptr); // Wyświetla wartość 10
```

Wskaźniki mogą wskazywać na tablice:

```
int tab[3] = {1, 2, 3};
int *p = tab;
printf("%d", *(p + 1)); // Wyświetla 2
```

### Arytmetyka wskaźników

Wskaźniki wspierają operacje matematyczne takie jak dodawanie, odejmowanie czy porównywanie.

### Dodawanie do wskaźnika

```
int arr[3] = {1, 2, 3};
int *p = arr;

printf("%d", *(p + 1)); // Wyświetla 2
```

### Porównywanie wskaźników

```
if (p < (p + 1)) {
    printf("p wskazuje na wcześniejszy element tablicy\n");
}
```

### Wskaźniki i tablice

Wskaźnik do tablicy przechowuje adres jej pierwszego elementu.

```
int arr[3] = {1, 2, 3};
int *p = arr;

printf("%d", *p); // Wyświetla 1
```

### Iteracja przez wskaźnik

```
for (int i = 0; i < 3; i++) {
    printf("%d ", *(p + i));
}
```

---

### Wskaźniki do wskaźników

Wskaźnik może przechowywać adres innego wskaźnika.

```
int x = 10;
int *p = &x;
int **pp = &p;

printf("%d", **pp); // Wyświetla 10
```

---

### Stałe i wskaźniki

**`const int *p`**

Wartość wskazywana przez wskaźnik jest stała, ale wskaźnik można zmienić.

```
const int x = 5;
const int *p = &x;
```

**`int *const p`**

Wskaźnik jest stały, ale wartość, na którą wskazuje, można zmienić.

```
int x = 5;
int *const p = &x;
*p = 10; // Poprawne
```

**`const int *const p`**

Ani wskaźnik, ani wartość, na którą wskazuje, nie mogą być zmieniane.

```
const int x = 5;
const int *const p = &x;
```

---

**Rozmiar wskaźnika:** Wskaźniki mają stały rozmiar na danej maszynie.

### Dynamiczna alokacja pamięci

Wskaźniki są kluczowe do dynamicznej alokacji pamięci w C.

```
int *p = (int *)malloc(10 * sizeof(int)); // Alokuje pamięć dla 10 elementów typu int
if (p != NULL) {
    for (int i = 0; i < 10; i++) {
        p[i] = i * 2;
    }
    free(p); // Zwalnia pamięć
}
```

---

### Wskaźniki do funkcji

Wskaźniki mogą przechowywać adresy funkcji, co umożliwia dynamiczne wywoływanie funkcji.

### Deklaracja wskaźnika do funkcji

```c
int add(int a, int b) {
    return a + b;
}

int (*funcPtr)(int, int) = &add;

printf("%d", funcPtr(2, 3)); // Wyświetla 5
```

---

### Typowe błędy z wskaźnikami

1. **Nieinicjalizowany wskaźnik**

   ```c
   int *p; // Brak inicjalizacji
   *p = 10; // Błąd: wskaźnik wskazuje na niezdefiniowany adres
   ```

2. **Zwolnienie już zwolnionej pamięci**

   ```c
   free(p);
   free(p); // Błąd: podwójne zwolnienie pamięci
   ```

3. **Dereferencja wskaźnika `NULL`** “‘c int $p = NULL;$ p = 10; // Błąd: brak alokacji pamięci

---

## 4. Struktury i dynamiczna alokacja pamięci

### Struktury

Struktury grupują różne typy danych:

```c
struct Vector {
    double x, y;
};

struct Vector v = {2.5, 3.5};
v.x = 5.0;
```

**struktury nie mągą zawierać zmiennej swojego typu (ale mogą wskaźnik)**

### Dynamiczna alokacja pamięci

Użycie funkcji takich jak `malloc` lub `calloc` umożliwia dynamiczne tworzenie struktur i tablic w czasie działania programu:

```
struct Vector *v = (struct Vector *)malloc(sizeof(struct Vector));
v->x = 2.5; // Użycie wskaźnika do modyfikacji struktury
(*v).y = 3.5; // Alternatywny zapis

free(v);
```

`realloc` służy do dynamicznej zmiany przydzielonej pamięci przy zachowaniu zawartości

```
int *ptr1, *ptr2, size;

// Allocate memory for four integers
size = 4 * sizeof(*ptr1);
ptr1 = malloc(size);

printf("%d bytes allocated at address %p \n", size, ptr1);

// Resize the memory to hold six integers
size = 6 * sizeof(*ptr1);
ptr2 = realloc(ptr1, size);

printf("%d bytes reallocated at address %p \n", size, ptr2);
```

Dynamiczne tablice dwuwymiarowe:

```
double (*arr2D)[N] = (double (*)[N])malloc(M * N * sizeof(double));
free(arr2D);
```

### Zmienne dynamiczne

Zmienne dynamiczne to te, których pamięć jest alokowana w czasie działania programu za pomocą funkcji takich jak `malloc`, `calloc` lub `realloc`. Są one zwalniane za pomocą `free`.

---

## 5. Funkcje i rekurencja

### Funkcje

Funkcje w C mogą zwracać wskaźniki i przyjmować wskaźniki jako argumenty:

```c
int *f(int *p) {
    *p = 8;
    return p;
}
```

### Funkcje do obsługi stringów

Wymagają dołączenia nagłówka `#include <string.h>`.

| Funkcja | Opis |
|---|---|
| `strlen(str)` | Długość stringa |
| `strcat(s1, s2)` | Łączy `s2` z `s1` |
| `strcpy(s1, s2)` | Kopiuje `s2` do `s1` |
| `strcmp(s1, s2)` | Porównuje stringi (0 = równe) |
| `strstr(s1, s2)` | s1 poszukiwany łańcuch znaków, s2 szukane znaki. szuka s2 w s1 i zwraca adres pierwszego wystąpienia |

### Rekurencja

Funkcja może wywoływać samą siebie:

```c
long factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

---

## 6. Obsługa plików

Pliki w C otwierane są za pomocą funkcji `fopen` i zamykane za pomocą `fclose`:

```c
FILE *fptr = fopen("plik.txt", "r");
if (fptr) {
    char ch;
    while ((ch = fgetc(fptr)) != EOF) {
        putchar(ch);
    }
    fclose(fptr);
}
```

### Otwieranie pliku

```c
FILE *fptr = fopen("plik.txt", "r");
```

| Tryb | Opis |
|---|---|
| `w` | Zapis |
| `a` | Dopisanie |
| `r` | Odczyt |

**Zamykanie pliku**

```
fclose(fptr);
```

---

## 7. Argumenty funkcji `main`

Funkcja `main` może przyjmować argumenty z wiersza poleceń:

```
int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

- `argc` - liczba argumentów (łącznie z nazwą programu).
- `argv` - tablica wskaźników do argumentów w postaci stringów.

Wywołanie programu:

```
program.exe 1 2 3
```

Wyświetli:

```
program.exe
1
2
3
```

---

## 8. Obsługa łańcuchów znaków

Do obsługi łańcuchów znaków w C używa się funkcji z biblioteki `string.h`:

```
#include <string.h>

const char *s1 = "Pierwszy kot, drugi kot, trzeci kot";
char s2[10] = "kot";
char *wsk = strstr(s1, s2);
printf("%s", wsk); // Wyświetli: "kot, drugi kot, trzeci kot"
```

### Enumy

```
enum Level {
    LOW,
    MEDIUM,
    HIGH
};

enum Level myLevel = MEDIUM;
```