Klasy zmiennych

Każda zmienna należy nie tylko do jakiegoś **typu**, ale także do określonej **klasy**.

Istnieją cztery słowa kluczowe opisujące **klasy zmiennych**: **extern** (ang. *external* - zewnętrzna), **auto** (ang. *automatic* - automatyczna), **static** (statyczna) oraz **register** (rejestrowa).

Zmienne zadeklarowane w obrębie funkcji należą standardowo do klasy auto, chyba że deklaracja stwierdza inaczej.

Argumenty formalne przekazywane do funkcji należą zazwyczaj również do klasy auto, ale mogą również należeć do klasy register.

Klasa zmiennej jest określona przez położenie deklaracji oraz użyte w niej słowo kluczowe.

Klasa określa trzy rzeczy:

I. decyduje ona o tym, które funkcje w programie mają dostęp do zmiennej. Jeśli fragment kodu może korzystać z określonej zmiennej, mówimy, że zmienna ta jest widoczna w tym fragmencie. Widoczność zmiennej w różnych częściach programu określa jej **zasięg** (ang. *scope*).

II. przesądza o tym, w ilu różnych miejscach może zostać zadeklarowana ta sama zmienna. Własność ta nosi nazwę **łączności** (ang. *linkage*).

III. określa, jak długo zmienna jest obecna w pamięci - czyli **czas trwania zmiennej**.

Zasięg

Zmienną w języku C charakteryzuje jeden z następujących trzech rodzajów zasięgu: **zasięg plikowy**, **zasięg blokowy** lub **zasięg prototypowy**.

Zmienna o **zasięgu plikowym** jest widoczna od miejsca jej definicji aż do końca pliku, w którym została zdefiniowana.

Zmienna o **zasięgu blokowym** jest widoczna od miejsca jej definicji do końca bloku zawierającego definicję. (blokiem nazywamy część programu ograniczoną klamrami.) Zasięg blokowy mają wszystkie zmienne lokalne, włącznie z argumentami formalnymi funkcji. Uznaje się, że blokiem zawierającym argumenty formalne funkcji jest treść funkcji.

Zasięg prototypowy odnosi się do nazw zmiennych użytych w prototypach funkcji, takich jak poniższy:

```
int mocarz(int mysz, double duży);
```

Zasięg prototypowy obejmuje obszar od definicji zmiennej do końca prototypu. Oznacza to po prostu, że kompilator interesują wyłącznie typy zmiennych zadeklarowanych w prototypie; użyte nazwy (jeśli w ogóle użyto nazw) nie mają znaczenia.

Łączność

Zmienną w języku C cechuje jeden z następujących typów łączności: **łączność zewnętrzna** (ang. *external linkage*), **łączność wewnętrzna** (ang. *internat linkage*) lub **brak łączności**.

Zmienna o **łączności zewnętrznej** może zostać użyta w każdym pliku źródłowym należącym do programu.

Zmienna o **łączności wewnętrznej** jest widoczna w tym pliku źródłowym, w którym została zdefiniowana.

Zmienna **pozbawiona łączności** jest widoczna tylko w swoim bloku.

Zmienne o łączności wewnętrznej lub zewnętrznej mogą występować w więcej niż jednej deklaracji, natomiast zmienne pozbawione łączności mogą być deklarowane tylko raz.

Zmienne o zasięgu blokowym lub prototypowym nie posiadają łączności. Zmienne o zasięgu plikowym może cechować łączność zewnętrzna lub wewnętrzna.

Czas trwania

Zmienną w języku C można określić za pomocą jednego z następujących dwóch **czasów trwania**: **statycznego** lub **automatycznego**.

Zmienna o **statycznym czasie trwania** istnieje przez cały czas działania programu. Przykładem takiej zmiennej jest każda zmienna zewnętrzna.

Zmienna o **automatycznym czasie trwania** istnieje tylko w czasie, kiedy program wykonuje blok, w którym została zdefiniowana. W tej kategorii mieszczą się wszystkie zmienne lokalne.

Zmienne automatyczne

Zmienne zadeklarowane w ramach funkcji należą do klasy automatycznej. Dla zwiększenia czytelności, można dodać do deklaracji słowo kluczowe **auto** tak, jak poniżej:

```
int main(void)
  {
  auto int plox;
```

Słowa kluczowego **auto** można użyć np. aby zaznaczyć, że z premedytacją dublujesz deklarację zmiennej zewnętrznej, lub że jest szczególnie istotne, aby klasa zmiennej nie została zmieniona.

Zmienna automatyczna ma zasięg blokowy. Tylko funkcja, w której zmienna została zadeklarowana, może uzyskać do niej dostęp przez podanie jej nazwy. Inna funkcja może korzystać ze zmiennej o tej samej nazwie, ale będzie ona wówczas niezależnym obiektem przechowywanym w innym miejscu pamięci.

Zmienna automatyczna nie posiada łączności. Nie można zadeklarować jej dwukrotnie w tym samym bloku. Możliwe jest zadeklarowanie dwóch zmiennych o tej samej nazwie w różnych blokach, jednak w takim wypadku są one oddzielnymi, nie powiązanymi ze sobą zmiennymi

Zmienna automatyczna jest zmienną o automatycznym czasie trwania. Zostaje ona utworzona w momencie wywołania zawierającej ją funkcji. Gdy funkcja zakończy działanie, a program powróci do funkcji wywołującej, zmienna automatyczna znika, a zajmowane przez nią miejsce jest udostępniane do użytku innych zmiennych.

W przypadku dwukrotnej deklaracji zmiennej o tej samej nazwie (raz w bloku podrzędnym, drugi raz poza nim), w ramach bloku podrzędnego widoczna jest zmienna, która została tam zdefiniowana. Mówimy, że **deklaracja wewnętrzna przesłania deklarację zewnętrzną**. Po wyjściu programu z bloku widoczna staje się zmienna zadeklarowana poza nim.

```
/* przesłona.c – zmienne w blokach */
#include <stdio.h>
int main()
  {
  int x = 30;
  printf("x w bloku zewnętrznym: %d\n", x);
    {
    int x = 77; // nowa zmienna x, przesiania pierwsza zmienna
    printf("x w bloku wewnętrznym: %d\n", x);
    }
  printf("x w bloku zewnętrznym: %d\n", x);
  while (x++ < 33)
    {
    int x = 100; // nowa zmienna x, przesiania pierwsza zmienna
    x++;
    printf("x w pętli while: %d\n", x};
    }
  return 0;
  }
```

```
Dane wyjściowe:
x w bloku zewnętrznym: 30
x w bloku wewnętrznym: 77
x w bloku zewnętrznym: 30
x w pętli while: 101
x w pętli while: 101
x w pętli while: 101
```

Inicjalizacja zmiennych automatycznych

Zmienne automatyczne nie otrzymują żadnej wartości początkowej, chyba że zostanie ona określona przez programistę.

```
int main(void)
{
int erpol;
int namioty = 5;
```

Wartością zmiennej `erpol` jest cokolwiek, co znajdowało się wcześniej w zajmowanych przez nią komórkach pamięci.

Zmienne zewnętrzne

Zmienną zewnętrzną jest każda zmienna zdefiniowana poza funkcją. Dla potrzeb dokumentacji zmienna zewnętrzna może również zostać zadeklarowana w obrębie wykorzystującej ją funkcji za pomocą słowa kluczowego **extern**.

Jeśli zmienna jest zdefiniowana w innym pliku, zadeklarowanie jej przy pomocy słowa **extern** jest obowiązkowe.

Deklaracje zmiennych zewnętrznych:

```
int Erview;      // zmienna zdefiniowana zewnętrznie
double Up[100];  // zmienna zdefiniowana zewnętrznie
extern char Kot; // obowiązkowa deklaracja; zmienna Kot jest zdefiniowana w innym pliku
void nast(void);
int main(void)
  {
  extern int Erview;  // opcjonalna deklaracja
  extern double Up[]; // opcjonalna deklaracja (rozmiar już został podany wyzej)
  . . .
  }
```

Zmienne zewnętrzne cechuje statyczny czas trwania. Tablica `Up` zachowuje swoje wartości niezależnie od tego, czy program wykonuje `main()`, `nast()`, czy jakakolwiek inną funkcję.

```
/* Przykład 1 */
int Hokus;
int magic();
int main(void)
  {
  extern int Hokus;
  . . .
  }
int magic ()
  {
  extern int Hokus;
  . . .
  }
```

```
/* Przykład 2 */
int Hokus;
int magic();
int main(void)
  {
  extern int Hokus;
  . . .
  }
int magic()
  {
  . . .
  }
```

```
/* Przykład 3 */
int Hokus;
int magic();
int main(void)
  {
  int Hokus; // przyłoniety zewn. Hokus
  . . .
  }
int Pokus;   // deklaracja zm. zewn. Pokus
int magic()
  {
  auto int Hokus // przyłoniety zewn. Hokus
  . . .
  }
```

```
. . . widoczna zmienna zewn. Hokus

. . . widoczna zmienna zewn. Pokus
```

UWAGA: Użycie słowa **extern** bez podania typu jest interpretowane jako zadeklarowanie typu **extern int**.

Inicjalizacja zmiennych zewnętrznych

W odróżnieniu od zmiennych automatycznych, zmienne zewnętrzne otrzymują w przypadku braku jawnej inicjalizacji domyślną wartość początkową 0. Zasada ta stosuje się również do elementów tablic zewnętrznych.

Definicje i deklaracje

Istnieje różnica między zdefiniowaniem zmiennej a jej zadeklarowaniem.

```
int erfejs; // definicja erfejs
main()
  {
  extern int erfejs;    // wykorzystanie zmiennej erfejs zdefiniowanej gdzie indziej
```

Pierwszą deklarację nazywamy **deklaracją definiującą** (ang. *defining declaration*). Powoduje ona zarezerwowanie w pamięci miejsca dla zmiennej.

Druga deklaracja, **deklaracja nawiązująca** (ang. *referencing declaration*) - wskazuje jedynie, że kompilator powinien skorzystać z utworzonej wcześniej zmiennej `erfejs` - nie jest więc ona definicją. Deklarację nawiązującą można rozpoznać po obecności słowa kluczowego **extern**; słowo to nakazuje ono kompilatorowi poszukiwanie definicji w innym miejscu programu.

W przypadku kodu:

```
extern int erfejs;
int main(void)
  {
```

kompilator przyjmie, że definicja zmiennej `erfejs` znajduje się w innym miejscu programu, być może w innym pliku źródłowym. Powyższa deklaracja nie powoduje przydzielenia pamięci.

Zmienna zewnętrzna może zostać zainicjalizowana tylko jeden raz, a inicjalizacja musi zostać dokonana razem z definicją.

```
extern char zezwól = 'Y';      // błąd
```

. . . ponieważ obecność słowa kluczowego `extern` sygnalizuje deklarację nawiązującą, a nie deklarację definiującą.

Zmienne statyczne

Określenie **zmienna statyczna** oznacza, że zmienna pozostaje w jednym miejscu w pamięci.

Zmienne statyczne mają taki sam zasięg, jak zmienne automatyczne, ale nie znikają, gdy zawierająca je funkcja zakończy działanie.

Zmienne należące do **klasy statycznej** charakteryzuje zasięg blokowy i statyczny czas trwania. Komputer przechowuje ich wartości między kolejnymi wywołaniami funkcji.

```
/* static.c – korzystanie ze zmiennej statycznej */
#include <stdio.h>
void proba_stat(void);
int main(void)
  {
  int licz;
  for(licz = 1; licz <= 3; licz++)
    {
    printf ("Oto iteracja nr %d: ", licz);
    proba_stat(void);
    }
  return 0;
  }
void proba_stat(void)
  {
  int zniknij = 1;          // zmienna automatyczna
  static int zostań = 1;    // zmienna statyczna
  printf("zniknij = %d, zostań = %d\n", zniknij++, zostań++);
  }
```

```
wynik:
Oto iteracja nr 1: zniknij = 1, zostań = 1
Oto iteracja nr 2: zniknij = 1, zostań = 2
Oto iteracja nr 3: zniknij = 1, zostań = 3
```

Zmienne statyczne zewnętrzne

Zmienną statyczną można również zadeklarować poza wszystkimi funkcjami, tworząc **zmienną statyczną zewnętrzną**.

Zwykła **zmienna zewnętrzna** może być używana przez funkcje w każdym pliku programu (ma *łączność zewnętrzną*).

Zmienna statyczna zewnętrzna - tylko przez funkcje znajdujące się w tym samym pliku, co jej definicja (*ma łączność wewnętrzną*).

Zmienną statyczną zewnętrzną tworzymy umieszczając jej definicję poza wszystkimi funkcjami i dodając do niej słowo kluczowe **static**:

```
static int losx = 1;
int los()
  {
```

Wiele plików

Bardziej złożone programy w języku C często zajmują kilka oddzielnych plików źródłowych. Zdarza się, że pliki te muszą skorzystać ze wspólnej zmiennej zewnętrznej.

W języku C dokonuje się tego przez umieszczenie w jednym pliku ***deklaracji definiującej***, a w pozostałych plikach - ***deklaracji nawiązujących***.

Wszystkie deklaracje oprócz jednej (*definiującej*) powinny zawierać słowo kluczowe **extern**, przy czym inicjalizacja zmiennej może mieć miejsce tylko w deklaracji definiującej.

Zauważ, że zmienna zewnętrzna zdefiniowana w jednym pliku nie staje się automatycznie dostępna w drugim pliku. Drugi plik musi zawierać deklarację ze słowem **extern**.

Zasięg funkcji

Pojęcie klas odnosi się nie tylko do zmiennych, ale także do funkcji.

Każda funkcja należy do klasy **zewnętrznej** (domyślnie) lub **statyczne**j.

Funkcja zewnętrzna jest dostępna dla funkcji znajdujących się w innych plikach, podczas gdy

funkcja statyczna może być wykorzystywana tylko w obrębie pliku, zawierającego jej definicję.

Załóżmy, że mamy plik zawierający następujące deklaracje funkcji:

```
double gamma();           // standardowo zewnętrzna
static double beta();     // statyczna
extern double delta();
```

Funkcje `gamma()` i `delta()` - w przeciwieństwie do `beta()` - mogą być wykorzystywane przez funkcje w innych plikach należących do programu.

Ponieważ funkcja `beta()` jest związana z jednym konkretnym plikiem, inne pliki mogą zawierać osobną funkcję o tej samej nazwie.

Zalecane jest korzystanie ze słowa kluczowego **extern** przy deklaracji funkcji, której definicja znajduje się w innym pliku (kwestia czytelności)

Zmienne rejestrowe

Zmienne rejestrowe - *przy odrobinie szczęścia* - mogą być przechowywane w rejestrach procesora lub w najszybszej dostępnej pamięci, co pozwala odczytywać je i przetwarzać ze znacznie większą prędkością niż zmienne zwykłego typu.

Wszystkie inne własności zmiennych rejestrowych są takie same, jak zmiennych automatycznych.

Zmienne rejestrowe deklarujemy:

```
int main(void)
  {
  register int szybki;
```

Deklarowanie zmiennej rejestrowej ma bardziej charakter prośby niż kategorycznego rozkazu. Kompilator musi uwzględnić liczbę wolnych rejestrów lub dostępną ilość szybkiej pamięci – można więc nie otrzymać spełnienia żądania. W takim przypadku deklarowana zmienna staje się zwykłą zmienną automatyczną.

Można zażądać, aby zmiennymi rejestrowymi były parametry formalne funkcji.

```
void macho(register int n)
```

Liczba typów, jakie mogą należeć do klasy register, może być ograniczona.

W odniesieniu do zmiennych rejestrowych nie można stosować operatora **&**.

Rzut kością

Spróbujmy napisać symulację jednej z bardziej popularnych czynności o charakterze losowym - rzutu kością.

Chcemy więc uzyskać liczbę losową z przedziału od 1 do 6.

Funkcja **`rand()`** daje w wyniku liczbę całkowitą z przedziału od 0 do **`RAND_MAX`**;

stała **`RAND_MAX`** jest zdefiniowana w pliku **`stdlib.h`** i zwykle jest równa **`INT_MAX`** (największej możliwej liczbie typu **`int`**).

Musimy więc dokonać kilku poprawek.

Oto jedna z możliwości:

1. Bierzemy resztę z dzielenia otrzymanej liczby przez 6 (za pomocą operatora modulo). Otrzymujemy liczbę całkowitą z przedziału od 0 do 5. .
2. Dodajemy 1. Nowa liczba należy do zakresu od 1 do 6.
3. Aby uogólnić proces, zastępujemy liczbę 6 w punkcie 1 dowolną liczbą ścianek, jakiej chcemy użyć.

```
/* kości.c */
#include <stdlib.h> // dla funkcji rand()
int rzut(int ścianki)
  {
  int wynik;
  wynik = rand() % ścianki + 1;
  return wynik;
  }
```

```c
/* rzuty.c – wielokrotne rzuty koscia */
#include <stdio.h>
#include <stdlib.h> // dla srand()
#include <time.h>   // dla time()
extern int rzut(int);
int main{void)
  {
  int kości, licznik, wynik, ściany;
  srand((unsigned int) time(0));  // wybiera losowe ziarno
  printf("Podaj liczbę ścian każdej kostki, wpisz 0, aby zakończyć.\n");
  while (scanf("%d", &sciany) == 1 && ściany > 0)
    {
    printf("Ile kosci?\n");
    scanf("%d", &kości);
    for (wynik = 0, licznik = 0; licznik < kości; licznik++) wynik += rzut(ściany);
    printf("Wyrzuciłeś %d na %d %d-sciennych kostkach.\n", wynik, kości, ściany);
    printf("Ile ścian? Wpisz 0, aby zakończyć.\n");
    }
  printf("ŻYCZĘ DUŻO SZCZĘŚCIA!\n");
  return 0;
  }
```

Przykład danych wyjściowych:

```
Podaj liczbę ścian każdej kostki, wpisz 0, aby zakończyć.
6
Ile kości?
2
Wyrzuciłeś 10 na 2 6-sciennych kostkach.
Ile ścian? Wpisz 0, aby zakończyć.
0
ZYCZE DUŻO SZCZĘŚCIA!
```

Kwalifikatory typów

Zmienną można opisać za pomocą jej **typu** oraz **klasy**.

Standard ANSI C dodaje dwie własności: **stałość** (ang. *constancy*) i **ulotność** (ang. *volatility*). Własności te deklarujemy za pomocą słów kluczowych **const** i **volatile**; słowa te tworzą tzw. **typy kwalifikowane** (ang. *qualified types*).

Kwalifikator const

Słowo kluczowe **const** tworzy zmienną, której wartość nie może zostać zmieniona za pomocą przypisania, inkrementacji ani dekrementacji.

W każdym kompilatorze zgodnym z ANSI kod

```
const int niezmien; // kwalifikuje niezmien jako stałą
niezmien = 12;      // niedozwolone
```

spowoduje wyświetlenie komunikatu obłędzie.

Zmienną zadeklarowaną jako **const** można zainicjować.

```
const int niezmien = 12; // ok
```

Powyższa deklaracja czyni z `niezmien` zmienną przeznaczoną tylko do odczytu.

Słowa kluczowego **const** można użyć do utworzenia tablicy, której program nie będzie w stanie zmienić:

```
const int dni[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Kwalifikator const a wskaźniki

W przypadku wskaźników, kwalifikator **const** może odnosić się zarówno do samego wskaźnika, jak i do wskazywanej przezeń wartości.

```
const float *wf;      // wf wskazuje na stałą wartość typu float
```

stwierdza, że `wf` wskazuje na wartość, która musi pozostać stała. Natomiast wartość samej zmiennej wskaźnikowej `wf` może ulec zmianie - można na przykład sprawić, aby wskazywała ona na inną wartość.

```
float *const wsk;       // wsk jest stałym wskaźnikiem
```

stwierdza, że sam wskaźnik `wsk` ma stałą wartość. Musi on wskazywać zawsze na to samo miejsce, ale znajdująca się w tym miejscu wartość może ulegać zmianie.

```
const float *const wsk;
```

oznacza, że `wsk` musi wskazywać zawsze na to samo miejsce oraz że zapisana w tym miejscu wartość nie może zostać zmieniona.

Kwalifikator const i deklaracje parametrów

Częstym zastosowaniem słowa **const** jest deklarowanie wskaźników będących formalnymi parametrami funkcji.

Ogólnie, przekazanie wskaźnika pozwala funkcji modyfikować dane w funkcji wywołującej, ale w celu uniknięcia tego używamy kwalifikatora **const**:

```
void wyświetl(const int tablica[], int granica);
```

Deklaracja `const int tablica[]` jest równoważna `const int *tablica`, a więc obie oznaczają, że dane wskazywane przez parametr tablica nie mogą ulec zmianie.

Kwalifikator const a dane globalne

Korzystanie ze zmiennych globalnych jest uważane za ryzykowne, ponieważ naraża ono dane na możliwość przypadkowej modyfikacji ze strony dowolnej części programu.

Ryzyko to znika, jeśli dane są stałe; używanie zmiennych globalnych posiadających kwalifikator **const**. Możliwe jest tworzenie stałych zmiennych, stałych tablic oraz stałych struktur.

Zagadnieniem, które wymaga uwagi, jest dzielenie tych samych stałych przez kilka plików. Istnieją tutaj dwie strategie:

I. stosowanie zwykłych zasad dla zmiennych zewnętrznych, czyli umieszczeniu deklaracji definiującej w jednym pliku i deklaracji nawiązujących (ze słowem **extern**) w pozostałych plikach.

```
/* pliki.c – definiuje kilka stałych globalnych */
const double PI = 3.14159;

/* plik2.c – korzysta ze stałych globalnych zdefiniowanych gdzie indziej */
extern const double PI;
```

II. umieszczenie stałych w pliku dołączanym. W tym przypadku należy pamiętać o dodatkowym kroku, jakim jest użycie klasy statycznej:

```
/* stale.h - definiuje kilka stałych globalnych */
static const double PI = 3.14159;

/* plik1.c – korzysta ze stałych globalnych zdefiniowanych gdzie indziej */
#include "stale.h"

/* plik2.c – korzysta ze stałych globalnych zdefiniowanych gdzie indziej */
#include "stale.h"
```

Jeśli nie użyjesz słowa **static**, dołączenie pliku *stale.h* do plików *plik1.c* i *plik2.c* spowoduje, że każdy z tych plików będzie zawierał **deklarację definiującą** tej samej zmiennej, co nie jest dozwolone. Nadanie każdej zmiennej **klasy statycznej zewnętrzne**j jest równoznaczne z przyznaniem każdemu plikowi osobnej kopii danych.

Kwalifikator volatile

Kwalifikator **volatile** informuje kompilator, że zmienna może zostać zmodyfikowana przez czynniki inne niż sam program.

Zwykle jest on wykorzystywany w odniesieniu do adresów sprzętowych lub danych użytkowanych wspólnie z innymi, działającymi równolegle programami.

Na przykład, wskaźnik może przechowywać adres aktualnego czasu systemowego. Wartość pod tym adresem ulega zmianie w czasie, niezależnie od tego, co robi program.

Taka sama sytuacja ma miejsce w przypadku danych odbieranych za pośrednictwem sieci komputerowej.

Składnia deklaracji:

```
volatile int ml;      // ml jest wartością ulotną
volatile int *wm      // wm wskazuje na wartość ulotną
```

Wartość może być równocześnie stała i ulotna.

Na przykład, zegar sprzętowy komputera nie powinien być modyfikowany przez program (czyli jest stały), ale równocześnie jest modyfikowany przez czynniki inne niż program (czyli jest ulotny).

Aby zadeklarować taką wartość, w deklaracji wystarczy użyć obu słów: **const** i **volatile**. Kolejność nie ma znaczenia.

```
volatile const int ml;
const volatile int *wm;
```

Struktury i inne formy danych

```
/* książka.c – spis jednej książki */
#include <stdio.h>
#define MAXTYT 41      // maksymalna dlugosc tytułu +1
#define MAXAUT 31      // maksymalna dlugosc nazwiska autora +1
struct książka         // szablon struktury o nazwie "książka"
   {
   char tytuł[MAXTYT];
   char autor[MAXAUT];
   float wartość;
   };                  // koniec szablonu struktury
int main(void)
   {
   struct książka bibl;   // declaracja bibl jako zmiennej typu książka
   printf("Podaj tytuł książki.\n");
   gets(bibl.tytuł);      // dostęp do składnika "tytuł"
   printf("Teraz podaj autora.\n");
   gets(bibl.autor);
   printf("Teraz podaj wartość.\n");
   scanf("%f", &bibl.wartość);
   printf("%s: \"%s\" (%.2f zl)\n", bibl.autor, bibl.tytuł, bibl.wartość);
   return 0;
   }
```

Deklaracja struktury

Deklaracja struktury jest planem, który opisuje budowę struktury.

```
struct książka
{
char tytuł[MAXTYT];
char autor[MAXAUT};
float wartość;
}
```

Deklaracja ta opisuje strukturę złożoną z dwóch tablic znakowych i jednej zmiennej typu float.

Nie tworzy ona rzeczywistego obiektu w pamięci, a jedynie określa, z czego składa się taki obiekt.

Na początku deklaracji znajduje się słowo kluczowe **struct**. Wskazuje ono, że to, co po nim następuje, jest strukturą.

Kolejnym elementem jest opcjonalna etykieta `książka` - jest ona nazwą przyporządkowaną strukturze.

Następnym elementem deklaracji struktury jest lista składników zawarta w klamrach.

Każdy składnik jest opisany przez swoją własną deklarację zakończoną średnikiem. Na przykład, składnik tytuł jest tablicą typu char posiadającą MAXTYT elementów. Składnik może należeć do dowolnego typu danych; może być nawet strukturą!

Definicję budowy struktury kończy klamra zamykająca i średnik.

Deklaracja struktury może zostać umieszczona poza wszystkimi funkcjami (zewnętrznie) lub w ramach definicji funkcji. W tym drugim przypadku, etykieta jest dostępna tylko w funkcji, w której znajduje się deklaracja.

Etykieta nie jest elementem obowiązkowym, ale jej użycie jest konieczne w sytuacji, kiedy szablon struktury jest zdefiniowany w jednym miejscu, a oparte na nim zmienne - w drugim.

Definiowanie zmiennej strukturalnej

Słowo **struktura** jest używane w dwóch znaczeniach.

Pierwszym z nich jest „plan strukturalny". Plan strukturalny informuje kompilator o tym, w jaki sposób mają zostać przedstawione dane, ale nie powoduje przydzielenia tym danym miejsca w pamięci.

Kolejnym krokiem jest utworzenie „zmiennej strukturalnej" lub „struktury" w drugim znaczeniu tego słowa. W naszym programie odpowiedzialny jest za to następujący wiersz:

```
struct książka bibl;
```

W odpowiedzi na tę instrukcję kompilator tworzy zmienną bibl. Zgodnie z szablonem książka rezerwuje on miejsce dla tablicy typu char zawierającej MAXTYT elementów, takiej samej tablicy zawierającej MAXAUT elementów oraz dla zmiennej typu float. Wszystkie te dane zostają połączone w jeden obiekt o nazwie bibl.

W deklaracji zmiennej strukturalnej `struct książka` pełni dokładnie tą samą rolą, co `int` lub `float` w zwykłych deklaracjach. Na przykład, możliwe jest zadeklarowanie dwóch zmiennych typu `struct książka` na raz lub nawet wskaźnika do tej struktury:

```
struct książka doyle, chandler, *wskks;
```

Z punktu widzenia komputera deklaracja

```
struct książka bibl;
```

jest skróconą formą deklaracji

```
struct książka // w tym przypadku etykietę ”książka” można pominąć
  {
  char tytuł[MAXTYT];
  char autor[MAXAUT];
  float wartość;
  }  bibl;    // po deklaracji następuje nazwa zmiennej
```

Inicjalizacja struktury

Aby zainicjalizować strukturę korzystamy ze składni podobnej do tej, którą stosujemy w przypadku tablic: korzystamy z ujętej w klamry listy wartości rozdzielonych przecinkami.

Każda wartość powinna należeć do tego samego typu, co odpowiadający jej składnik struktury.

```
struct book bibl = {"Pirat i dziewica", "Rene Vivotte", 7.95};
```

Uzyskiwanie dostępu do składników struktury

Poszczególne składniki struktury wskazujemy przy pomocy kropki (.) - operatora przynależności do struktury.

Na przykład, `bibl.wartość` oznacza pole `wartość` struktury `bibl`.

Z wyrażenia `bibl.wartość` można korzystać dokładnie w taki sam sposób, jak z każdej innej zmiennej typu `float`.

Podobnie wyrażenie `bibl.tytuł` jest traktowane tak samo, jak zwykła tablica typu `char`.

```
gets(bibl.tytuł);
scanf("%f", &bibl.wartość);
```

Deklarowanie tablicy struktur

Deklaracja tablicy struktur jest taka sama, jak w przypadku każdej innej tablicy:

```
struct książka bibl[MAXKS];
```

Powyższa instrukcja stwierdza, że `bibl` jest tablicą złożoną z `MAXKS` elementów. Każdy element jest strukturą typu `książka`.

Wskazywanie składników tablicy struktur

Wskazując składniki tablicy struktur, stosujemy te same zasady, co w przypadku pojedynczej struktury: Do nazwy struktury dodajemy nazwę składnika poprzedzoną kropką.

```
bibl[0].wartość  // wartość przechowywana w pierwszym elemencie tablicy
bibl[4].tytuł    // tytuł przechowywany w piątym elemencie tablicy
```

Zauważ, że indeks tablicy znajduje się przed kropką, a nie na końcu wyrażenia:

```
bibl.wartość[2]  // ŹLE
bibl[2].wartość  // DOBRZE
```

Struktury zagnieżdżone

```
struct daneos // pierwszy szablon
  {
  char imię[DL];
  char nazw[DL];
  };
struct facet // drugi szablon (+ deklaracja!)
  {
  struct daneos person;  // struktura zagnieżdżona
  char ulub_jedz[DL];
  char zawód[DL];
  float dochody;
  } gosc;
```

Dostęp:

```
gosc.person.imię
gosc.dochody
```

Wskaźniki do struktur

```
/* znajom.c – wykorzystuje wskaźnik do struktury */
#include <stdio.h>
#define DL 20

struct daneos {char imię[DL];char nazw[DL]; };
struct facet  {struct daneos person;char ulub_jedz[DL];char zawód[DL];float dochody; };

int main(void)
{
struct facet gosc[2] = {{{ "Chip", "Hyperlink"}, "chipsy", "makler pamięciowy", 36827.00},
                        {{"Norbert", "Brzuchacz"}, "łosoś", "redaktor brukowca", 148500.00}}
struct facet *on;     // deklaracja wskaźnika do struktury

on = &gosc[0];        // inicjacja wskaźnika do struktury
printf("on->dochody ma wartość %.2f $: (*on).dochody ma wartość %.2f $\n", on->dochody, (*on).dochody);
on++;                 // wskazanie na następna strukturę
printf("on->ulub_jedz ma wartość %s: on->person.nazw ma wartość %s\n", on->ulub_jedz, on->person.nazw);
return 0;
}
```

```
dane wyjściowe:
on->dochody ma wartość 36827.00 $:   (*on).dochody ma wartość 36827.00 $
on->ulub_jedz ma wartość łosoś:    on->person.nazw ma wartość Brzuchacz
```

Dostęp do składników za pomocą wskaźnika

Wskaźnik do struktury z operatorem -> działa tak samo, jak nazwa struktury z operatorem . (kropką).

Jeśli `on == &gosc[0]`

`on->dochody == gosc[0].dochody`

Jeśli `on == &gosc[0]`, to `*on == gosc[0]` więc

`gosc[0].dochody == (*on).dochody`

Nawiasy są wymagane, ponieważ operator . ma wyższy priorytet niż *.

Podsumowując, jeśli `on == &gosc[0]`

`gosc[0].dochody == (*on).dochody == on -> dochody`

Struktury a funkcje

Kompilatory oferują wybór między przekazaniem do funkcji całej struktury lub jej adresu; a w przypadku, gdy istotna jest tylko część struktury, można również przekazać pojedynczy składnik.

Przekazanie składników struktur

```
/* fundl.c – przekazywanie składników struktury jako argumentów */
#include <stdio.h>
#define FUNDDL 50
struct fundusze{char  bank[FUNDDL]; double bankfund; char oszcz[FUNDDL]; double oszczfund;};
double suma(double, double);
int main(void)
  {
  struct fundusze edek = {"Bank", 2024.72, "Kasa”, 8237.11};
  printf("Edek posiada w sumie %.2f zl.\n", suma(edek.bankfund, edek.oszczfund));
  return 0;
  }
double suma(double x, double y)
  {
  return(x + y) ;
  }
```

Przekazanie wskaźnika do struktury

```
/* fund2.c – przekazywanie wskaźnika do struktury */
#include <stdio.h>
#define FUNDDL 50
struct fundusze{char  bank[FUNDDL]; double bankfund; char oszcz[FUNDDL]; double oszczfund;};
double suma(const struct fundusze *);    // argument jest wskaźnikiem
int main(void)
  {
  struct fundusze edek = {"Bank", 2024.72, "Kasa”, 8237.11};
  printf("Edek posiada w sumie %.2f zl.\n", suma(&edek));
  return 0;
  }
double suma(const struct fundusze *pieniądze)
  {
  return(pieniadze->bankfund + pieniadze->oszczfund);
  }
```

Przekazanie struktury jako argumentu

```
/* fund3.c – przekazywanie struktury */
#include <stdio.h>
#define FUNDDL 50
struct fundusze{char  bank[FUNDDL]; double bankfund; char oszcz[FUNDDL]; double oszczfund;};
double suma(struct fundusze mamona);  // argumentem jest struktura
int main(void)
  {
  struct fundusze edek = {"Bank", 2024.72, "Kasa”, 8237.11};
  printf("Edek posiada w sumie %.2f zl.\n", suma(edek));
  return 0;
  }
double suma (struct fundusze mamona)
  {
  return(mamona.bankfund + mamona.oszczfund);
  }
```

Uwaga:

Przy wywołaniu funkcji `suma()` utworzona zostaje zmienna automatyczna `mamona` oparta na szablonie `fundusze`. Następnie składniki struktury `mamona` otrzymują wartości odpowiadających im składników w strukturze `edek`. Tym samym, obliczenia dokonywane są na kopii pierwotnej struktury, podczas gdy w programie `fund2.c` wykonywane były na oryginale.

Struktury jako argumenty i wartości funkcji

```
/* imienaz2.c – przekazuje i zwraca struktury */
#include <stdio.h>
#include <string.h>
struct daneos   // definicja struktury
  {
  char imię[20];
  char nazw[20];
  int litery;
  };

// prototypy
struct daneos pobierz(void);
struct daneos oblicz(struct daneos);
void pokaz(struct daneos);

int main(void)
  {
  struct daneos osoba; // deklaracja
  osoba = pobierz();
  osoba = oblicz(osoba);
  pokaz(osoba);
  return 0;
  }
// definicje funkcji
struct daneos pobierz(void)
  {
  struct daneos temp;
  printf("Podaj swoje imie.\n");
  gets(temp.imię);
  printf("Podaj swoje nazwisko.\n");
  gets (temp.nazw);
  return temp;
  }
struct daneos oblicz(struct daneos info)
  {
  info.litery = strlen(info.imię)
                + strlen(info.nazw);
  return info;
  }
void pokaz(struct daneos info)
  {
  printf("%s %s, Twoje imię i nazwisko składają ");
  printf("sie z %d liter.\n", info.imię,
                        info.nazw, info.litery);
  }
```

Zapisywanie zawartości struktury w pliku

Zawartość struktury można zapisywać do pliku składnik po składniku używając funkcji fprintf(). Metoda ta staje się dość nieporęczna w przypadku struktur zawierających np. 30 składników. Ponadto, stwarza ona problem przy odczytywaniu, ponieważ program musiałby wiedzieć, gdzie kończy się jedno pole, a zaczyna drugie. Problem ten można wyeliminować przez użycie formatu o stałej szerokości pola, jednak wówczas trzeba pamiętać o określeniu pól o wystarczającym rozmiarze.

Lepszym rozwiązaniem jest użycie funkcji `fread()` i `fwrite()` do odczytu i zapisu jednostek o rozmiarze struktury.

Funkcje te zapisują i odczytują informacje korzystając z tej samej binarnej reprezentacji, której używa program.

Instrukcja

```
fwrite(&slownik, sizeof(struct książka), 1, pksiazka);
```

kopiuje całość struktury `słownik` do pliku związanego ze wskaźnikiem plikowym `pksiazka`.

Argument `sizeof(struct książka)` informuje funkcję o długości bloku, a liczba 1 wskazuje, że należy skopiować tylko jeden blok.

Funkcja `fread()` w przypadku przekazania jej tych samych argumentów kopiuje porcję danych o rozmiarze struktury z pliku pod adres `kslownik`.

```
fread(&slownik, sizeof(struct książka), 1, pksiazka);
```

Unie

Unia (ang. *union*) jest typem, który pozwala przechowywać różne rodzaje danych w tym samym obszarze pamięci (jednak nie równocześnie).

Dzięki **uniom** możliwe jest na przykład utworzenie tablicy jednostek o jednakowej długości, z których każda może przechowywać dane innego typu.

Tworzenie **unii** przebiega podobnie, jak tworzenie struktury. Wyróżniamy **szablony** i **zmienne**, które mogą zostać zdefiniowane w jednym lub - za pośrednictwem **etykiety** - w dwóch krokach.

Przykład szablonu unii z etykietą:

```
union magazyn
  {
  int    cyfra;
  double duzfl;
  char   litera;
  }
```

Poniższe instrukcje definiują trzy unie typu magazyn:

```
union magazyn fit;      // unia typu magazyn
union magazyn tab[10];  // tablica 10 unii typu magazyn
union magazyn *wu;      // wskaźnik do unii typu magazyn
```

Deklaracja **unii** powoduje, że kompilator przydziela jej tyle (i tylko tyle !) pamięci,
aby mogła ona przechować największą ze zmiennych będących częścią szablonu unii.

Unie mogą być inicjalizowane; ponieważ jednak każda unia przechowuje tylko jedną wartość, zasady są nieco inne niż w przypadku struktur. Istnieją dwie możliwości: nadanie unii wartości innej unii tego samego typu lub inicjalizacja pierwszego składnika unii:

```
union magazyn wartA;
wartA.litera =  'R';
union magazyn wartB = wartA;      // przypisanie wartości innej unii
union magazyn wartC = {88};       // inicjalizacja składnika cyfra
```

Przykłady wykorzystania unii:

```
fit.cyfra = 23;      //              23 zapisane w fit;   zajęte 2 bajty
fit.duzfl = 2.0;     // 23 usunięte,  2.0 zapisane;        zajęte 8 bajtów
fit.litera = 'h';    // 2.0 usunięte, h zapisane;          zajęty 1 bajt
```

Pamiętanie o tym, jaki rodzaj informacji znajduje się w unii w każdym momencie, jest obowiązkiem programisty !

Dobrym miejscem do zastosowania unii może być struktura, w której rodzaj przechowywanych danych zależy od jednego ze składników.

Na przykład, załóżmy, że mamy strukturę reprezentującą samochód. Jeśli właścicielem samochodu jest jego użytkownik, potrzebny jest składnik struktury opisujący osobę właściciela. Jeśli zaś samochód jest wypożyczony, składnik powinien opisywać firmę, która jest właścicielem pojazdu. Aby to uzyskać, możemy użyć następującego kodu:

```
struct wlasc
  {
  char nrdowodu[12]
  . . .
  };
struct firma
  {
  char nazwa[40];
  char siedziba[40]
  . . .
  };
union dane
  {
  struct właściciel wlasc_sam;
  struct firma firma_sam;
  };
struct dane_sam
  {
  char marka[15];
  int stan;   // 0 = własny, 1 = pożyczony
  union dane dane wlasc;
  . . .
  }
```

typedef

Słowo kluczowe **typedef** jest zaawansowanym elementem języka C, który pozwala tworzyć nowe nazwy typów. Przypomina ono dyrektywę #define z trzema istotnymi różnicami:

I. **typedef** nadaje nazwy typom, a nie wartościom.

II. Słowo kluczowe **typedef** jest interpretowane przez kompilator, a nie preprocesor.

III. W swoim obszarze zastosowań mechanizm **typedef** jest bardziej elastyczny niż **#define**.

Załóżmy, że chcemy używać terminu BYTE w odniesieniu do liczb o rozmiarze jednego bajta. W tym celu wystarczy zdefiniować BYTE jak zwykłą zmienną typu char i poprzedzić definicję słowem kluczowym typedef.

```
typedef unsigned char BYTE;
```

Od tego momentu możemy korzystać z nowego typu BYTE przy deklarowaniu zmiennych:

```
BYTE x, y[10], *z;
```

zamiast

```
unsigned char x, y[10], *z;
```

Zasięg definicji typu zależy od położenia instrukcji **typedef**. Jeśli definicja znajduje się wewnątrz funkcji jej zasięg jest lokalny; jeśli znajduje się ona poza funkcją, jej zasięg jest globalny.

Inny przykład:

```
typedef struct{float re; float im}  complex;

complex a, b, z = {1.0, 2.0};  // deklaracja zmiennych zespolonych (z = 1 + i2)
```

deklaracje

Znaczenie deklaracji może zostać zmienione przez dodanie do nazwy zmiennej tzw. modyfikatora.

Modyfikator	***Znaczenie***
*	wskaźnik
()	funkcja
[]	tablica

Możliwe jest użycie wielu modyfikatorów w jednej deklaracji, co pozwala tworzyć duży zakres różnych typów:

```
int plansza[8][8];     // tablica tablic typu int
int **wsk;             // wskaźnik do wskaźnika do int
int *domy[10];         // 10-cio elementowa tablica wskaźników do int
int (*domy)[10];       // wskaźnik do 10-cio elementowej tablicy typu int
int *uff[3][4];        // tablica 3x4 wskaźników do int
int (*puff)[3][4];     // wskaźnik do tablicy 3x4 typu int
int (*huff[3])[4];     // 3-elementowa tablica wskaźników do 4-elementowych tablic typu int
char *flip();          // funkcja zwracająca wskaźnik do char
char (*flap)();        // wskaźnik do funkcji, która zwraca wartość typu char
char (*flop[3])();     // tablica 3 wskaźników do funkcji zwracających wartość typu char
```

Funkcje a wskaźniki

Wskaźnik do funkcji przechowuje adres, pod którym rozpoczyna się kod maszynowy funkcji.
Wskaźnik do funkcji, należy określić typ wskazywanej funkcji. Określenie typu funkcji odbywa się przez podanie typów argumentów oraz typu wartości zwracanej.

Na przykład, mamy prototyp:

```
void DuzeLit(char *);  // przetwarza małe litery na duże
```

Aby zadeklarować wskaźnik wf do funkcji tego typu, należy użyć następującej instrukcji:

```
void (*wf)(char *);    // wf jest wskaźnikiem do funkcji
```

Pierwsza para nawiasów związuje operator * z nazwą `wf`, co oznacza, że `wf` jest wskaźnikiem do funkcji.
Wyrażenie `(*wf)` jest funkcją o argumencie typu `(char *)` i wartości zwracanej typu `void`.

Najprostszym sposobem na zapamiętanie tej deklaracji jest zauważenie, że wyrażenie `(*wf)` zastępuje w niej nazwę funkcji `DuzeLit`.

Po utworzeniu wskaźnika do funkcji, może on otrzymywać adresy funkcji odpowiedniego typu. Adres funkcji symbolizuje w tym kontekście jej nazwa:

```
void DuzeLit(char *);  // prototyp
void MaleLit(char *);  // prototyp
int zaokr(double);     // prototyp
void (*wf)(char *);    // deklaracja wskaźnika
wf = DuzeLit;     // prawidłowe, DuzeLit jest adresem funkcji
wf = MaleLit;     // prawidłowe, MaleLit jest adresem funkcji
wf = zaokr;       // nieprawidłowe, zaokr jest funkcją niewłaściwego typu
wf = MaleLit();   // nieprawidłowe, MaleLit() nie jest adresem
```

Tak jak wskaźnik do danych pozwala uzyskać dostęp do wskazywanych wartości, tak i wskaźnik do funkcji umożliwia skorzystanie ze wskazywanej funkcji.
Istnieją dwie równoprawne, ale sprzeczne logicznie składnie, pozwalające to uczynić:

```
void DuzeLit(char *);  // prototyp
void MaleLit(char *);  // prototyp
void (*wf)(char *);    // deklaracja wskaźnika
char mis[] = "Nina Metier";
wf = DuzeLit;    // wf wskazuje na funkcję DuzeLit, wyrażenie *wf oznacza funkcję DuzeLit
(*wf)(mis);      // . . . więc (*wf) (mis) to tyle, co DuzeLit (mis)
wf = MaleLit;    // nazwa funkcji jest wskaźnikiem, więc nazwy i wskaźnika można używać zamiennie
wf(mis);         // . . . więc wf (mis) jest równoważne MaleLit (mis)
```

Podobnie jak wskaźniki do danych, wskaźniki do funkcji są najczęściej wykorzystywane w roli argumentów.

```
void pokaz(void (*fw)(char *), char *lan);
```

Funkcja `pokaz()` może wywołać wskazywaną funkcję za pomocą jednego z dwóch zapisów - `fw()` lub `(*fw)()`.

```
void pokaz(void (*fw)(char *), char *lan)
  {
  (*fw)(lan);    // stosuje wybraną funkcję do lan
  puts(lan);     // wyświetla wynik
  }
```

Zastowanie instrukcji typedef

```
typedef void (*WdF)(char *); // "definicja" typu: wskaźnik do funkcji
                             // . . . i zastosowanie do . . .
void pokaz(WdF wp, char *);  // deklaracji argumentu (zamiast void (*wp)(char *) )
WdF wfun;                    // deklaracji wskaźnika (zamiast void (*wfun)(char *) )
WdF tabwf[4] = {DuzeLit,  MaleLit,  Odwroc}; // deklaracji tablicy wskaźników
```

Podsumowanie wszystkich pięciu zastosowań nazw funkcji.

nazwa funkcji w prototypie:	`int licz(int x, int y);`
nazwa funkcji w definicji:	`int licz(int x, int y) { . . . }`
nazwa funkcji w wywołaniu:	`stan = licz(q, r);`
nazwa funkcji jako wskaźnik:	`wfun = licz;`
nazwa funkcji jako argument wskaźnikowy:	`slowsort(tab, n, licz);`

□