Podstawy Programowania

Wykład VII

Zmienne dynamiczne, struktury, moduły programowe

Robert Muszyński
Katedra Cybernetyki i Robotyki, PWr

Zagadnienia: zmienne dynamiczne, tablice dynamiczne, struktury, wskaźniki do struktur, struktury zawierające tablice, tablice zawierające struktury, rozdzielna kompilacja – moduły.

– Skład FoilTEX –

Zmienne dynamiczne

Dostępna dla programu pamięć komputera, dzieli się na cztery obszary:

- **kod programu**,

Zmienne dynamiczne

Dostępna dla programu pamięć komputera, dzieli się na cztery obszary:

- **kod programu**,
- **dane statyczne** (np. stałe i zmienne globalne programu),

Zmienne dynamiczne

Dostępna dla programu pamięć komputera, dzieli się na cztery obszary:

- **kod programu**,
- **dane statyczne** (np. stałe i zmienne globalne programu),
- **dane automatyczne** – zmienne tworzone i usuwane automatycznie w trakcie działania programu na tzw. stosie (np. zmienne lokalne funkcji),

Zmienne dynamiczne

Dostępna dla programu pamięć komputera, dzieli się na cztery obszary:

- **kod programu**,
- **dane statyczne** (np. stałe i zmienne globalne programu),
- **dane automatyczne** – zmienne tworzone i usuwane automatycznie w trakcie działania programu na tzw. stosie (np. zmienne lokalne funkcji),
- **dane dynamiczne** – organizowane przez menadżera pamięci dynamicznej na tzw. stercie, tworzone i usuwane na żądanie w dowolnym momencie pracy programu.

Zmienne dynamiczne cd.

Zmienne dynamiczne:

- są **anonimowe**, tzn. nie mają nazwy – dostęp do takich zmiennych możliwy jest jedynie poprzez ich adres w pamięci (poprzez zmienne wskaźnikowe),

Zmienne dynamiczne cd.

Zmienne dynamiczne:

- są **anonimowe**, tzn. nie mają nazwy – dostęp do takich zmiennych możliwy jest jedynie poprzez ich adres w pamięci (poprzez zmienne wskaźnikowe),
- muszą być **tworzone jawnie** przez programistę (rezerwacja pamięci),

Zmienne dynamiczne cd.

Zmienne dynamiczne:

- są **anonimowe**, tzn. nie mają nazwy – dostęp do takich zmiennych możliwy jest jedynie poprzez ich adres w pamięci (poprzez zmienne wskaźnikowe),
- muszą być **tworzone jawnie** przez programistę (rezerwacja pamięci),
- mają **nieograniczony zakres**, tzn. można odowływać sie do nich z dowlonego miejsca programu,

Zmienne dynamiczne cd.

Zmienne dynamiczne:

- są **anonimowe**, tzn. nie mają nazwy – dostęp do takich zmiennych możliwy jest jedynie poprzez ich adres w pamięci (poprzez zmienne wskaźnikowe),
- muszą być **tworzone jawnie** przez programistę (rezerwacja pamięci),
- mają **nieograniczony zakres**, tzn. można odowływać sie do nich z dowlonego miejsca programu,
- **istnieją dowolnie długo** tzn. do chwili ich usunięcie przez programistę (zwalnianie pamięci),

Zmienne dynamiczne cd.

Zmienne dynamiczne:

- są **anonimowe**, tzn. nie mają nazwy – dostęp do takich zmiennych możliwy jest jedynie poprzez ich adres w pamięci (poprzez zmienne wskaźnikowe),
- muszą być **tworzone jawnie** przez programistę (rezerwacja pamięci),
- mają **nieograniczony zakres**, tzn. można odowływać sie do nich z dowlonego miejsca programu,
- **istnieją dowolnie długo** tzn. do chwili ich usunięcie przez programistę (zwalnianie pamięci),
- tak samo jak dla zmiennych statycznych obowiązuje w ich przypadku ścisłe przestrzeganie **zgodności typów**,

Zmienne dynamiczne cd.

Zmienne dynamiczne:

- są **anonimowe**, tzn. nie mają nazwy – dostęp do takich zmiennych możliwy jest jedynie poprzez ich adres w pamięci (poprzez zmienne wskaźnikowe),
- muszą być **tworzone jawnie** przez programistę (rezerwacja pamięci),
- mają **nieograniczony zakres**, tzn. można odowływać sie do nich z dowlonego miejsca programu,
- **istnieją dowolnie długo** tzn. do chwili ich usunięcie przez programistę (zwalnianie pamięci),
- tak samo jak dla zmiennych statycznych obowiązuje w ich przypadku ścisłe przestrzeganie **zgodności typów**,
- korzystanie z nieprzydzielonego we właściwy sposób obszaru pamięci **najprawdopodobniej spowoduje błąd**,

Zmienne dynamiczne cd.

Zmienne dynamiczne:

- są **anonimowe**, tzn. nie mają nazwy – dostęp do takich zmiennych możliwy jest jedynie poprzez ich adres w pamięci (poprzez zmienne wskaźnikowe),
- muszą być **tworzone jawnie** przez programistę (rezerwacja pamięci),
- mają **nieograniczony zakres**, tzn. można odowływać sie do nich z dowlonego miejsca programu,
- **istnieją dowolnie długo** tzn. do chwili ich usunięcie przez programistę (zwalnianie pamięci),
- tak samo jak dla zmiennych statycznych obowiązuje w ich przypadku ścisłe przestrzeganie **zgodności typów**,
- korzystanie z nieprzydzielonego we właściwy sposób obszaru pamięci **najprawdopodobniej spowoduje błąd**,
- próba zwolnienia nieprzydzielonego obszaru pamięci **powoduje nieprzewidziane działanie**.

Dynamiczne przydzielanie pamięci

W języku C do dynamicznego przydzielania i zwalniania pamięci służą funkcje z nagłówka `stdlib.h`:

- `void *malloc(size_t n)` zwraca wskaźnik do `n` bajtów niezainicjowanej pamięci albo `NULL`,

Dynamiczne przydzielanie pamięci

W języku C do dynamicznego przydzielania i zwalniania pamięci służą funkcje z nagłówka `stdlib.h`:

- `void *malloc(size_t n)` zwraca wskaźnik do `n` bajtów niezainicjowanej pamięci albo `NULL`,
- `void *calloc(size_t n, size_t rozmiar)` zwraca wskaźnik do obszaru mogącego pomieścić `n` elementów podanego rozmiaru `rozmiar` zainicjowanego zerami albo `NULL`,

Dynamiczne przydzielanie pamięci

W języku C do dynamicznego przydzielania i zwalniania pamięci służą funkcje z nagłówka `stdlib.h`:

- `void *malloc(size_t n)` zwraca wskaźnik do `n` bajtów niezainicjowanej pamięci albo `NULL`,
- `void *calloc(size_t n, size_t rozmiar)` zwraca wskaźnik do obszaru mogącego pomieścić `n` elementów podanego rozmiaru `rozmiar` zainicjowanego zerami albo `NULL`,
- `void free(void *wskaznik)` zwalnia pamięć wskazywaną przez `wskaznik`, przy czym wartość wskaźnika `wskaznik` musi być wynikiem wcześniejszego wywołania funkcji `malloc` lub `calloc`.

Dynamiczne przydzielanie pamięci

W języku C do dynamicznego przydzielania i zwalniania pamięci służą funkcje z nagłówka `stdlib.h`:

- `void *malloc(size_t n)` zwraca wskaźnik do `n` bajtów niezainicjowanej pamięci albo `NULL`,
- `void *calloc(size_t n, size_t rozmiar)` zwraca wskaźnik do obszaru mogącego pomieścić `n` elementów podanego rozmiaru `rozmiar` zainicjowanego zerami albo `NULL`,
- `void free(void *wskaznik)` zwalnia pamięć wskazywaną przez `wskaznik`, przy czym wartość wskaźnika `wskaznik` musi być wynikiem wcześniejszego wywołania funkcji `malloc` lub `calloc`.

```
int *wsk;
wsk = (int*) malloc(sizeof(int)); /* przydzielenie pamieci */
if(wsk == NULL) {                 /* sprawdzamy czy sie udalo */
  printf ("blad przydzialu pamieci\n"); exit(-1);
}
*wsk = 17; *wsk *=3;  /* dzialania na zmiennej dynamicznej */
free(wsk);           /* zwalniamy pamiec przed zakonczeniem */
```

Dynamiczne przydzielanie pamięci cd.

Prościutkie przykłady:

```
int  *i1, *i2;

i1 = (int*) malloc(sizeof(int));
*i1 = 5;
i2 = (int*) malloc(sizeof(int));
*i2 = 5;
if (i1 == i2) printf("Wskazniki takie same.\n");
else printf("Wskazniki rozne.\n");

i2 = i1;   /* !!! ZGUBIONA ZMIENNA, NIEUZYTEK !!! */
if (i1 == i2) printf("Wskazniki takie same.\n");
else printf("Wskazniki rozne.\n");

*i2 = 7
if (*i1 == *i2) printf("Wartosci takie same.\n");
else printf("Wartosci rozne.\n");
```

Zmienne dynamiczne – tablica jednowymiarowa

Przykład operowania na dynamicznej tablicy jednowymiarowej:

```
void main(void) {
  int rozmiar_tablicy, i;
  double *tablica_liczb;









}
```

Zmienne dynamiczne – tablica jednowymiarowa

Przykład operowania na dynamicznej tablicy jednowymiarowej:

```
void main(void) {
  int rozmiar_tablicy, i;
  double *tablica_liczb;
  printf("Ile liczb chcesz wprowadzic: ");
  scanf("%d", &rozmiar_tablicy);
  if( tablica_liczb =
  (double*) calloc(rozmiar_tablicy, sizeof(double)))









}
```

Zmienne dynamiczne – tablica jednowymiarowa

Przykład operowania na dynamicznej tablicy jednowymiarowej:

```
void main(void) {
  int rozmiar_tablicy, i;
  double *tablica_liczb;
  printf("Ile liczb chcesz wprowadzic: ");
  scanf("%d", &rozmiar_tablicy);
  if( tablica_liczb =
  (double*) calloc(rozmiar_tablicy, sizeof(double)))
  {
    for(i = 0; i < rozmiar_tablicy; i++)
      tablica_liczb[i] = 100;
                          /*alt. *(tablica_liczb + i) = 100;*/


  }
}
```

Zmienne dynamiczne – tablica jednowymiarowa

Przykład operowania na dynamicznej tablicy jednowymiarowej:

```
void main(void) {
  int rozmiar_tablicy, i;
  double *tablica_liczb;
  printf("Ile liczb chcesz wprowadzic: ");
  scanf("%d", &rozmiar_tablicy);
  if( tablica_liczb =
  (double*) calloc(rozmiar_tablicy, sizeof(double)))
  {
    for(i = 0; i < rozmiar_tablicy; i++)
      tablica_liczb[i] = 100;
                          /*alt. *(tablica_liczb + i) = 100;*/
  ...
  free(tablica_liczb);
  }
}
```

Tablica statyczna versus dynamiczna

Utworzenie statycznej tablicy jednowymiarowej:

```
#define ROZMIAR_TABLICY 100
double tablica_statyczna[ROZMIAR_TABLICY];
```

Tablica statyczna versus dynamiczna

Utworzenie statycznej tablicy jednowymiarowej:

```
#define ROZMIAR_TABLICY 100
double tablica_statyczna[ROZMIAR_TABLICY];
```

Utworzenie dynamicznej tablicy jednowymiarowej:

```
int rozmiar_tablicy=7;
double *tablica_dynamiczna;

tablica_dynamiczna =
        (double*) calloc(rozmiar_tablicy, sizeof(double));
```

Tablica statyczna versus dynamiczna

Utworzenie statycznej tablicy jednowymiarowej:

```
#define ROZMIAR_TABLICY 100
double tablica_statyczna[ROZMIAR_TABLICY];
```

Utworzenie dynamicznej tablicy jednowymiarowej:

```
int rozmiar_tablicy=7;
double *tablica_dynamiczna;

tablica_dynamiczna =
        (double*) calloc(rozmiar_tablicy, sizeof(double));
```

Odwoływanie się do tablic:

```
int i = 3;

tablica_statyczna[i] = 4;
tablica_dynamiczna[i] = 4;
```

Dynamiczna tablica dwuwymiarowa

Nie ma prostego sposobu zdefiniowania dynamicznej tablicy dwuwymiarowej.

Dynamiczna tablica dwuwymiarowa

Nie ma prostego sposobu zdefiniowania dynamicznej tablicy dwuwymiarowej.

```
int main() {
  int wym_x, wym_y;              /* wymiary tablicy */

  wym_x = 5; wym_y = 7;          /* symulujemy pobranie wymiarow */









  return 0;
}
```

Dynamiczna tablica dwuwymiarowa

Nie ma prostego sposobu zdefiniowania dynamicznej tablicy dwuwymiarowej.

```
int main() {
  int wym_x, wym_y;              /* wymiary tablicy */

  wym_x = 5; wym_y = 7;          /* symulujemy pobranie wymiarow */

     /* definiujemy zmienna wskaznikowa odpowiedniej postaci */
  int (*tablica)[wym_x];
     /* i powolujemy nasza tablice do zycia */
  tablica = (int(*)[wym_x]) malloc(wym_x*wym_y*sizeof(int));









  return 0;
}
```

Dynamiczna tablica dwuwymiarowa

Nie ma prostego sposobu zdefiniowania dynamicznej tablicy dwuwymiarowej.

```
int main() {
  int wym_x, wym_y;              /* wymiary tablicy */

  wym_x = 5; wym_y = 7;          /* symulujemy pobranie wymiarow */
     /* definiujemy zmienna wskaznikowa odpowiedniej postaci */
  int (*tablica)[wym_x];
     /* i powolujemy nasza tablice do zycia */
  tablica = (int(*)[wym_x]) malloc(wym_x*wym_y*sizeof(int));

    /* teraz mozemy robic z nia co chcemy :) */
  for(int i = 0; i < wym_y; i++)
    for(int j = 0; j < wym_x; j++)
      tablica[i][j] = 10*i+j;
  return 0;
}
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                         lista_deklaracji_skladowych "}"
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;
struct {
  int re;
  int im;
} zm2, zm3;
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;

struct {
  int re;
  int im;
} zm2, zm3;

zm1.re = 10;
zm1.im = 20;
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;

struct {
  int re;
  int im;
} zm2, zm3;

zm1.re = 10;
zm1.im = 20;
```

```
struct zsp {
  int re;
  int im;
} zm1;
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;

struct {
  int re;
  int im;
} zm2, zm3;

zm1.re = 10;
zm1.im = 20;
```

```
struct zsp {
  int re;
  int im;
} zm1;

struct zsp zm2,zm3;
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;
struct {
  int re;
  int im;
} zm2, zm3;

zm1.re = 10;
zm1.im = 20;
```

```
struct zsp {
  int re;
  int im;
} zm1;
struct zsp zm2,zm3;




zm2.re = zm1.re + 10;
zm2.im = zm1.im + 15;
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;
struct {
  int re;
  int im;
} zm2, zm3;

zm1.re = 10;
zm1.im = 20;
```

```
struct zsp {
  int re;
  int im;
} zm1;
struct zsp zm2,zm3;




zm2.re = zm1.re + 10;
zm2.im = zm1.im + 15;
```

```
struct zsp {
  int re;
  int im;
};
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;
struct {
  int re;
  int im;
} zm2, zm3;

zm1.re = 10;
zm1.im = 20;
```

```
struct zsp {
  int re;
  int im;
} zm1;
struct zsp zm2,zm3;



zm2.re = zm1.re + 10;
zm2.im = zm1.im + 15;
```

```
struct zsp {
  int re;
  int im;
};
struct zsp zm1;
struct zsp zm2,zm3;
```

Struktury

Składnia specyfikatora struktury zapisana w notacji MBNF

```
specyfikator_struktury = "struct" [ identyfikator ] "{"
                          lista_deklaracji_skladowych "}"
                        | "struct" identyfikator .
```

Przykłady:

```
struct {
  int re;
  int im;
} zm1;
struct {
  int re;
  int im;
} zm2, zm3;

zm1.re = 10;
zm1.im = 20;
```

```
struct zsp {
  int re;
  int im;
} zm1;
struct zsp zm2,zm3;




zm2.re = zm1.re + 10;
zm2.im = zm1.im + 15;
```

```
struct zsp {
  int re;
  int im;
};
struct zsp zm1;
struct zsp zm2,zm3;



zm3.re = zm1.re;
zm3.im = zm2.re;
```

Operowanie strukturami

Dozwolonymi opercjami dla struktury są:

- przypisanie innej struktury w całości
- skopiowanie jej w całości na inną strukturę

Operowanie strukturami

Dozwolonymi opercjami dla struktury są:

- przypisanie innej struktury w całości
- skopiowanie jej w całości na inną strukturę
- pobranie jej adresu za pomocą operatora referencji &

Operowanie strukturami

Dozwolonymi opercjami dla struktury są:

- przypisanie innej struktury w całości
- skopiowanie jej w całości na inną strukturę
- pobranie jej adresu za pomocą operatora referencji &
- odwołanie się do jej składowych

Operowanie strukturami

Dozwolonymi opercjami dla struktury są:

- przypisanie innej struktury w całości
- skopiowanie jej w całości na inną strukturę
- pobranie jej adresu za pomocą operatora referencji `&`
- odwołanie się do jej składowych
- przez kopiowanie i przypisanie rozumie się także przesyłanie argumentów funkcjom i zwracanie przez funkcje wartości.

Operowanie strukturami – przykłady

```
struct zsp zrob_zesp(int x, int y) {
  struct zsp temp;

  temp.re = x; temp.im = y;
  return temp
}
```

Operowanie strukturami – przykłady

```
struct zsp zrob_zesp(int x, int y) {
  struct zsp temp;

  temp.re = x; temp.im = y;
  return temp
}
            /* i gdzies dalej w programie */
l1 = zrob_zesp(0,0);
l2 = zrob_zesp(a,b);
```

Operowanie strukturami – przykłady

```
struct zsp zrob_zesp(int x, int y) {
  struct zsp temp;

  temp.re = x; temp.im = y;
  return temp
}
            /* i gdzies dalej w programie */
l1 = zrob_zesp(0,0);
l2 = zrob_zesp(a,b);
```

```
struct zsp dodaj(struc zsp x, struct zsp y)
{
  x.re += y.re;
  x.im += y.im;
  return x;
}
```

Operowanie strukturami – przykłady

```
struct zsp zrob_zesp(int x, int y) {
  struct zsp temp;

  temp.re = x; temp.im = y;
  return temp
}
            /* i gdzies dalej w programie */
l1 = zrob_zesp(0,0);
l2 = zrob_zesp(a,b);
```

```
struct zsp dodaj(struc zsp x, struct zsp y)
{
  x.re += y.re;
  x.im += y.im;
  return x;
}
            /* i gdzies dalej w programie */
l1 = dodaj(l1, l2);
```

Operowanie strukturami – typedef

Mieliśmy

```
struct zsp {
  int re;
  int im;
};
```

Operowanie strukturami – typedef

Mieliśmy

```
struct zsp {
  int re;
  int im;
};
struct zsp zm1;
struct zsp zm2,zm3;
```

Operowanie strukturami – typedef

Mieliśmy

```
struct zsp {
  int re;
  int im;
};
struct zsp zm1;
struct zsp zm2,zm3;
```

A możemy napisać

```
typedef struct {
  int re;
  int im;
} zespolona;
```

Operowanie strukturami – typedef

Mieliśmy

```
struct zsp {
  int re;
  int im;
};
struct zsp zm1;
struct zsp zm2,zm3;
```

A możemy napisać

```
typedef struct {
  int re;
  int im;
} zespolona;
zespolona zm1;
zespolona zm2,zm3;
```

Operowanie strukturami – typedef

Mieliśmy

```
struct zsp {
  int re;
  int im;
};
struct zsp zm1;
struct zsp zm2,zm3;
```

A możemy napisać

```
typedef struct {
  int re;
  int im;
} zespolona;
zespolona zm1;
zespolona zm2,zm3;
```

i teraz

```
zespolona dodaj(zespolona x, zespolona y)
{
  x.re += y.re;
  x.im += y.im;
  return x;
}
```

Operowanie strukturami – wskaźniki

Zdefiniujmy

```
zespolona dodaj(zespolona *x, zespolona *y)
{
  (*x).re += (*y).re;
  (*x).im += (*y).im;
  return *x;
}
```

Operowanie strukturami – wskaźniki

Zdefiniujmy

```
zespolona dodaj(zespolona *x, zespolona *y)
{
  (*x).re += (*y).re;
  (*x).im += (*y).im;
  return *x;
}
            /* i gdzies dalej w programie */
l3 = dodaj(&l1, &l2);
```

Operowanie strukturami – wskaźniki

Zdefiniujmy

```
zespolona dodaj(zespolona *x, zespolona *y)
{
  (*x).re += (*y).re;
  (*x).im += (*y).im;
  return *x;
}
            /* i gdzies dalej w programie */
l3 = dodaj(&l1, &l2);
```

Co raczej zapisujemy przy użyciu operatora ->

```
zespolona dodaj(zespolona *x, zespolona *y)
{
  x->re += y->re;
  x->im += y->im;
  return *x;
}
```

Tablice a wskaźniki – przypomnienie

Pamiętamy, że tablice i wskaźniki mogą być inicjowane stałą wartością:

```
char tab[] = "To jest string.";
char *ptr = "Jak rowniez to.";
```

Tablice a wskaźniki – przypomnienie

Pamiętamy, że tablice i wskaźniki mogą być inicjowane stałą wartością:

```
char tab[] = "To jest string.";
char *ptr = "Jak rowniez to.";
```

Uzyskujemy w ten sposób dwie tablice znakowe, lecz poprzez istotnie różne zmienne. `tab` jest tablicą, której zawartość jest zainicjalizowana określonymi znakami, której nie można zmienić jako zmiennej, ale której wszystkie pozycje znakowe mogą być dowolnie zmieniane. Natomiast `ptr` jest zmienną wskaźnikową zainicjalizowaną wskaźnikiem na napis znakowy. Wartość tej zmiennej wskaźnikowej można zmieniać dowolnie, lecz zawartości pozycji znakowych nie (napis jest tablicą stałą, przydzieloną w pamięci stałych).

Tablice a wskaźniki – przypomnienie

Pamiętamy, że tablice i wskaźniki mogą być inicjowane stałą wartością:

```
char tab[] = "To jest string.";
char *ptr = "Jak rowniez to.";
```

Uzyskujemy w ten sposób dwie tablice znakowe, lecz poprzez istotnie różne zmienne. `tab` jest tablicą, której zawartość jest zainicjalizowana określonymi znakami, której nie można zmienić jako zmiennej, ale której wszystkie pozycje znakowe mogą być dowolnie zmieniane. Natomiast `ptr` jest zmienną wskaźnikową zainicjalizowaną wskaźnikiem na napis znakowy. Wartość tej zmiennej wskaźnikowej można zmieniać dowolnie, lecz zawartości pozycji znakowych nie (napis jest tablicą stałą, przydzieloną w pamięci stałych).

```
tab[1] = ptr[1];     /* poprawne kopiowanie znakow */
*(tab+1) = *(ptr+1); /* rowniez poprawne */
tab = ptr;           /* to przypisanie jest NIEDOZWOLONE */

ptr[1] = tab[1];     /* kopiowanie znakow NIEDOZWOLONE */
*(ptr+1) = *(tab+1); /* rowniez NIEDOZWOLONE */
ptr = tab;           /* poprawne, choc gubi pamiec */
```

Struktury zawierające tablice

Struktura dogodna dla obrazków i wczytywanie danych do niej ze strumienia

```
typedef struct {
  int wym_x, wym_y, odcieni;
  int piksele[1000][1000];
} t_obraz;
```

Struktury zawierające tablice

Struktura dogodna dla obrazków i wczytywanie danych do niej ze strumienia

```
typedef struct {
  int wym_x, wym_y, odcieni;
  int piksele[1000][1000];
} t_obraz;

int czytaj(FILE *plik_we, t_obraz *obraz) {
  ...
  fscanf(plik_we,"%d", &(obraz->wym_x)); /* analog wym_y */
  for(i = 0; i < obraz->wym_x; i++)
    for(j = 0; j < obraz->wym_y; j++)
      fscanf(plik_we, "%d", &(obraz->piksele[i][j]));
}
```

Struktury zawierające tablice

Struktura dogodna dla obrazków i wczytywanie danych do niej ze strumienia

```
typedef struct {
  int wym_x, wym_y, odcieni;
  int piksele[1000][1000];
} t_obraz;

int czytaj(FILE *plik_we, t_obraz *obraz) {
  ...
  fscanf(plik_we,"%d", &(obraz->wym_x)); /* analog wym_y */
  for(i = 0; i < obraz->wym_x; i++)
    for(j = 0; j < obraz->wym_y; j++)
      fscanf(plik_we, "%d", &(obraz->piksele[i][j]));
}
  t_obraz obrazek1;        /* i gdzies w jakies funkcji */
  czytaj(plik1, &obrazek1);
```

Struktury zawierające tablice

Struktura dogodna dla obrazków i wczytywanie danych do niej ze strumienia

```
typedef struct {
  int wym_x, wym_y, odcieni;
  int piksele[1000][1000];
} t_obraz;

int czytaj(FILE *plik_we, t_obraz *obraz) {
  ...
  fscanf(plik_we,"%d", &(obraz->wym_x)); /* analog wym_y */
  for(i = 0; i < obraz->wym_x; i++)
    for(j = 0; j < obraz->wym_y; j++)
      fscanf(plik_we, "%d", &(obraz->piksele[i][j]));
}
  t_obraz obrazek1;        /* i gdzies w jakies funkcji */
  czytaj(plik1, &obrazek1);

  t_obraz *obrazek1;                  /* albo dynamicznie */
  obrazek1 = (t_obraz *) malloc(sizeof(t_obraz));
  czytaj(plik1, obrazek1);
```

Struktury zawierające tablice cd.

To samo co poprzednio, ale z dynamicznie alokowaną tablicą na piksele!

```
typedef struct {
  int wym_x, wym_y, odcieni;
  void *piksele;              /* tu lezy pies pogrzebany, a czemu nie int */
} t_obraz;
```

Struktury zawierające tablice cd.

To samo co poprzednio, ale z dynamicznie alokowaną tablicą na piksele!

```
typedef struct {
  int wym_x, wym_y, odcieni;
  void *piksele;                /* tu lezy pies pogrzebany, a czemu nie int */
} t_obraz;

int czytaj(FILE *plik_we, t_obraz *obraz) {
  ...
  fscanf(plik_we,"%d", &(obraz->wym_x));  /* analogicznie wym_y */
```

Struktury zawierające tablice cd.

To samo co poprzednio, ale z dynamicznie alokowaną tablicą na piksele!

```
typedef struct {
  int wym_x, wym_y, odcieni;
  void *piksele;                /* tu lezy pies pogrzebany, a czemu nie int */
} t_obraz;

int czytaj(FILE *plik_we, t_obraz *obraz) {
  ...
  fscanf(plik_we,"%d", &(obraz->wym_x));  /* analogicznie wym_y */
                                  /* rezerwujemy odpowiednio duza tablice */
  obraz->piksele = malloc(obraz->wym_x*obraz->wym_y*sizeof(int));
```

Struktury zawierające tablice cd.

To samo co poprzednio, ale z dynamicznie alokowaną tablicą na piksele!

```
typedef struct {
  int wym_x, wym_y, odcieni;
  void *piksele;                /* tu lezy pies pogrzebany, a czemu nie int */
} t_obraz;

int czytaj(FILE *plik_we, t_obraz *obraz) {
  ...
  fscanf(plik_we,"%d", &(obraz->wym_x));  /* analogicznie wym_y */
                                  /* rezerwujemy odpowiednio duza tablice */
  obraz->piksele = malloc(obraz->wym_x*obraz->wym_y*sizeof(int));
  /* dopiero teraz definiujemy zmienna pomocnicza jako wskaznik na tablice */
  /* o znanych wymiarach, tylko przez nia bedziemy sie odwolywac do pola   */
  int (*piksele)[obraz->wym_x];          /* piksele w strukturze z obrazem */
```

Struktury zawierające tablice cd.

To samo co poprzednio, ale z dynamicznie alokowaną tablicą na piksele!

```
typedef struct {
  int wym_x, wym_y, odcieni;
  void *piksele;                /* tu lezy pies pogrzebany, a czemu nie int */
} t_obraz;

int czytaj(FILE *plik_we, t_obraz *obraz) {
  ...
  fscanf(plik_we,"%d", &(obraz->wym_x));  /* analogicznie wym_y */
                                  /* rezerwujemy odpowiednio duza tablice */
  obraz->piksele = malloc(obraz->wym_x*obraz->wym_y*sizeof(int));
  /* dopiero teraz definiujemy zmienna pomocnicza jako wskaznik na tablice */
  /* o znanych wymiarach, tylko przez nia bedziemy sie odwolywac do pola   */
  int (*piksele)[obraz->wym_x];          /* piksele w strukturze z obrazem */
  piksele=(int(*)[obraz->wym_x]) obraz->piksele; /*inicjujemy go jak trzeba*/
```

Struktury zawierające tablice cd.

To samo co poprzednio, ale z dynamicznie alokowaną tablicą na piksele!

```
typedef struct {
  int wym_x, wym_y, odcieni;
  void *piksele;                /* tu lezy pies pogrzebany, a czemu nie int */
} t_obraz;

int czytaj(FILE *plik_we, t_obraz *obraz) {
  ...
  fscanf(plik_we,"%d", &(obraz->wym_x));  /* analogicznie wym_y */
                                  /* rezerwujemy odpowiednio duza tablice */
  obraz->piksele = malloc(obraz->wym_x*obraz->wym_y*sizeof(int));
  /* dopiero teraz definiujemy zmienna pomocnicza jako wskaznik na tablice */
  /* o znanych wymiarach, tylko przez nia bedziemy sie odwolywac do pola   */
  int (*piksele)[obraz->wym_x];          /* piksele w strukturze z obrazem */

  piksele=(int(*)[obraz->wym_x]) obraz->piksele; /*inicjujemy go jak trzeba*/

  for(i = 0; i < obraz->wym_y; i++)             /* i dzialamy!!! */
    for(j = 0; j < obraz->wym_x; j++)
      fscanf(plik_we, "%d", &(piksele[i][j]));
}
```

Moduły programowe

- dołączanie plików poleceniem preprocesora `#include`

```
#include <stdio.h>

int Silnia(int N) {
  if (N < 0)
    printf("Funkcja: Silnia, blad:
                    ujemny argument: %d\n", N);
  else if (N == 0)
    return(1);
  else return(N * Silnia(N-1));
} /* Silnia */

int main() {
  int X;

  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

Moduły programowe

- dołączanie plików poleceniem preprocesora `#include`

```
#include <stdio.h>

int Silnia(int N) {
  if (N < 0)
    printf("Funkcja: Silnia, blad:
                    ujemny argument: %d\n", N);
  else if (N == 0)
    return(1);
  else return(N * Silnia(N-1));
} /* Silnia */

int main() {
  int X;

  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

⟹ **plik: silnia.c**

Moduły programowe

- dołączanie plików poleceniem preprocesora `#include`

```
#include <stdio.h>

int Silnia(int N) {
  if (N < 0)
    printf("Funkcja: Silnia, blad:
                    ujemny argument: %d\n", N);
  else if (N == 0)
    return(1);
  else return(N * Silnia(N-1));
} /* Silnia */
```

⟹ **plik: silnia.c**

```
int main() {
  int X;

  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

⟹ **plik: main.c**

plik: silnia.c

```
#include <stdio.h>

int Silnia(int N) {
  if (N < 0)
    printf("Funkcja: Silnia, blad: ujemny argument: %d\n", N);
  else if (N == 0)
    return(1);
  else return(N * Silnia(N-1));
} /* Silnia */
```

plik: silnia.c

```
#include <stdio.h>

int Silnia(int N) {
  if (N < 0)
    printf("Funkcja: Silnia, blad: ujemny argument: %d\n", N);
  else if (N == 0)
    return(1);
  else return(N * Silnia(N-1));
} /* Silnia */
```

plik: main.c

```
#include <stdio.h>


int main() {
  int X;
  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

plik: silnia.c

```
#include <stdio.h>

int Silnia(int N) {
  if (N < 0)
    printf("Funkcja: Silnia, blad: ujemny argument: %d\n", N);
  else if (N == 0)
    return(1);
  else return(N * Silnia(N-1));
} /* Silnia */
```

plik: main.c

```
#include <stdio.h>
#include "silnia.c"        /* N I E S T O S O W A N E */

int main() {
  int X;
  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

plik: silnia.c

```
#include <stdio.h>

int Silnia(int N) {
  if (N < 0)
    printf("Funkcja: Silnia, blad: ujemny argument: %d\n", N);
  else if (N == 0)
    return(1);
  else return(N * Silnia(N-1));
} /* Silnia */
```

plik: main.c

```
#include <stdio.h>
#include "silnia.c"        /* N I E S T O S O W A N E */

int main() {
  int X;
  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

Kompilacja

`diablo 21:cc -Xc main.c` $\Longleftarrow$ tworzy `a.out`

- tworzenie modułów

plik: modul.c

```
#include <stdio.h>
                                        /* identyczny jak silnia.c */
int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
} /* Silnia */
```

- tworzenie modułów

plik: modul.c

```
#include <stdio.h>
                                                    /* identyczny jak silnia.c */
int Silnia(int N) {
  if (N < 0)
    ...                              /* jak poprzednio */
} /* Silnia */
```

plik: main.c

```
#include <stdio.h>


int main() {
  int X;
  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

- tworzenie modułów

plik: modul.c

```
#include <stdio.h>
                                                    /* identyczny jak silnia.c */
int Silnia(int N) {
  if (N < 0)
    ...                              /* jak poprzednio */
} /* Silnia */
```

plik: main.c

```
#include <stdio.h>

int Silnia(int);                     /* prototyp funkcji importowanej */

int main() {
  int X;
  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

- tworzenie modułów

plik: modul.c

```
#include <stdio.h>
                                                  /* identyczny jak silnia.c */
int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
} /* Silnia */
```

plik: main.c

```
#include <stdio.h>

int Silnia(int);                 /* prototyp funkcji importowanej */

int main() {
  int X;
  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

Kompilacja (lub jak na stronie następnej)

`diablo 21:cc -Xc -c modul.c` $\Longleftarrow$ tworzy `modul.o`

- tworzenie modułów

plik: modul.c

```
#include <stdio.h>
                                        /* identyczny jak silnia.c */
int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
} /* Silnia */
```

plik: main.c

```
#include <stdio.h>

int Silnia(int);                 /* prototyp funkcji importowanej */

int main() {
  int X;
  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

Kompilacja (lub jak na stronie następnej)

```
diablo 21:cc -Xc -c modul.c   ⟸ tworzy modul.o
diablo 22:cc -Xc -c main.c    ⟸ tworzy main.o
```

- tworzenie modułów

plik: modul.c

```
#include <stdio.h>
                                        /* identyczny jak silnia.c */
int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
} /* Silnia */
```

plik: main.c

```
#include <stdio.h>

int Silnia(int);                 /* prototyp funkcji importowanej */

int main() {
  int X;
  printf("Podaj argument dla funkcji Silnia: ");
  scanf("%d",&X);
  printf("Silnia(%1d) = %d\n", X, Silnia(X));
}
```

Kompilacja (lub jak na stronie następnej)

```
diablo 21:cc -Xc -c modul.c   ⟸ tworzy modul.o
diablo 22:cc -Xc -c main.c    ⟸ tworzy main.o
diablo 23:cc main.o modul.o   ⟸ tworzy a.out
```

plik: modul.c

```
#include <stdio.h>


int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
```

plik: modul.c

```
#include <stdio.h>


int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
```

plik: modul.h

```
int Silnia(int);          /* prototyp funkcji eksportowanej */
```

plik: modul.c

```
#include <stdio.h>

int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
```

plik: modul.h

```
int Silnia(int);          /* prototyp funkcji eksportowanej */
```

plik: main.c

```
#include <stdio.h>

int main() {
    ...                          /* jak poprzednio */
}
```

plik: modul.c

```
#include <stdio.h>


int Silnia(int N) {
  if (N < 0)
    ...                        /* jak poprzednio */
```

plik: modul.h

```
int Silnia(int);          /* prototyp funkcji eksportowanej */
```

plik: main.c

```
#include <stdio.h>
#include "modul.h"

int main() {
    ...                        /* jak poprzednio */
}
```

plik: modul.c

```
#include <stdio.h>


int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
```

plik: modul.h

```
int Silnia(int);          /* prototyp funkcji eksportowanej */
```

plik: main.c

```
#include <stdio.h>
#include "modul.h"

int main() {
    ...                          /* jak poprzednio */
}
```

Kompilacja (lub jak na stronie poprzedniej)

`diablo 21:cc -Xc -c modul.c` ⟸ tworzy `modul.o`

plik: modul.c

```
#include <stdio.h>


int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
```

plik: modul.h

```
int Silnia(int);          /* prototyp funkcji eksportowanej */
```

plik: main.c

```
#include <stdio.h>
#include "modul.h"

int main() {
    ...                          /* jak poprzednio */
}
```

Kompilacja (lub jak na stronie poprzedniej)

```
diablo 21:cc -Xc -c modul.c          ⟸ tworzy modul.o
diablo 22:cc -Xc main.c modul.o      ⟸ tworzy a.out
```

plik: modul.c

```
#include <stdio.h>
#include "modul.h"

int Silnia(int N) {
  if (N < 0)
    ...                          /* jak poprzednio */
```

plik: modul.h

```
int Silnia(int);          /* prototyp funkcji eksportowanej */
```

plik: main.c

```
#include <stdio.h>
#include "modul.h"

int main() {
    ...                          /* jak poprzednio */
}
```

Kompilacja (lub jak na stronie poprzedniej)

```
diablo 21:cc -Xc -c modul.c        ⟸ tworzy modul.o
diablo 22:cc -Xc main.c modul.o    ⟸ tworzy a.out
```

Podsumowanie

- Zagadnienia podstawowe

1. Jak zadeklarować dynamicznie zmienną typu całkowitego?
2. W jaki sposób przydziela się pamięć dla zmiennych wskaźnikowych?
3. Do czego służą funkcje: `malloc`, `calloc`? Czym się różnią? Czym różnią się następujące instrukcje, jeśli chodzi o rezultaty:
   ```
   int * p1 = (int *) malloc (100 * sizeof(int))
   int * p1 = (int *) calloc (100, sizeof(int))?
   ```
4. Na czym polega i z czego wynika tworzenie nieużytków, jak temu zaradzić?
5. Czym różnią się dane statyczne od danych automatycznych?
6. Czym różnią się tablice statyczne od tablic dynamicznych?
7. Dlaczego nie jest możliwe skompilowanie programu zawierającego poniższe deklaracje globalne:
   ```
   int a = 10;
   int tab[a][a+1];?
   ```
8. Porównaj zalety tablic i struktur.
9. W celu skopiowania zawartości struktury zawierającej tablicę wystarczy użyć zwykłego przypisania? Jeśli tak, co zostanie skopiowane gdy tablica ta jest statyczna, a co gdy dynamiczna?
10. Podaj przykład deklaracji tablicy struktur oraz sposobu odwoływania się do elemwntów takiej tablicy.

11. Zaproponuj stukturę do przechowywania informacji o osobie (imię, nazwisko, rok urodzenia, adres (dodatkowa stuktura?) itp.)
12. Wymień zalety definiowania typów zmiennych (`typedef`).
13. Do czego służy pisanie programu w modułach?
14. Czy dzielenie programu na moduły ma na celu m.in. szybszą jego kompilację?
15. Co powinien zawierać plik źródłowy (`*.c`), a co nagłówkowy (`*.h`) w programie o budowie modułowej? Gdzie powinny się znaleźć warunki PRE i POST?
16. Jaka jest rola plików nagłówkowych?
17. Jak kompiluje się programy, w których skład wchodzą moduły?

• Zagadnienia rozszerzające

1. Wskaż różne sposoby zdefiniowania i wykorzystania dynamicznej tablicy dwuwymiarowej (w tym z wykorzystaniem makra pozwalającego na zdefiniowanie tablicy jednowymiarowej i odwoływanie się do niej jak do tablicy dwuwymiarowej).
2. W jaki sposób przy tworzeniu struktury typu `t_obraz` z dynamicznie alokowaną tablicą na piksele (pokazaną na slajdzie 15 można wykorzystać funkcję zdefiniowaną poniżej?

```
int (*alloc_tablica(int wymiar))[]
{
  int (*tablica)[wymiar];
  tablica = (int(*)[wymiar]) malloc(wymiar*wymiar*sizeof(int));
  return tablica;
}
```

W jaki sposób najlepiej zorganizować obsługę sytuacji, w ktorej tablica nie zostanie poprawnie zaalokowana (funkcja `malloc` zwróci wartość `NULL`)?

3. Czy standard ANSI C przewiduje możliwość przydzielenia konkretnego miejsca w pamięci?
4. Jakie są możliwe scenariusze w przypadku korzystania przez program z niezaalokowanej pamięci? Jakie rodzaje błędów mogą się pojawić?
5. Jaka jest różnica między strukturami (`struct`) a uniami (`union`)?
6. Czym są i jakie mają znaczenie kwalifikatory języka C typu `const`, `volatile` oraz `restrict`?
7. Wyjaśnij pojęcie struktur zagnieżdżonych.
8. Poczytaj o elastycznych składnikach tablicowych w odniesieniu do struktur.
9. Na czym polega automatyzacja procesu kompilacji za pomocą programu `make`? Napisz `makefile` dla swojego programu na przetwarzanie obrazów w wersji modułowej.

• Zadania

1. Przy użyciu funkcji `malloc` napisz własną funkcję równoważną w działaniu funkcji `calloc`.
2. Napisz funkcję, która pyta o imię, zapisuje je w utworzonej zmiennej dynamicznej i zwraca wskaźnik do niej. Przetestuj działanie funkcji, nie zapomnij o zwalnianiu pamięci przed zakończeniem głównego programu.
3. Napisz program wykonujący proste operacje na macierzach (dodawanie, mnożenie), przy czym macierze powinny być tworzone dynamicznie na podstawie wymiarów podanych

przez użytkownika i przechowywane w odpowiednich strukturach.

4. Załóżmy, że dana jest statyczna tablica dwuwymiarowa 7x15 liczb całkowitych. Zapisz program, który przekopiuje dane z tej tablicy do tablicy dynamicznej, a następnie wyświetli zawartość nowej tablicy. Dopisz do programu funkcję, która znajdzie i wyświetli element maksymalny.
5. Wykorzystaj tablicę dwuwymiarową do napisania programu obsługującego szachownicę, w którym można byłoby zdefiniować układ figur na planszy a następnie sprawdzić możliwość wykonania zadanego ruchu (lub wyświetlić wszystkie dozwolone ruchy dla zadanej figury).
6. Napisz funkcję odejmującą dwie liczby zespolone.
7. Zaproponuj strukturę danych do przechowywania informacji o produktach (nazwa, ilość, cena). Napisz program, który pobierze dane o dwóch produktach i porówna je ze sobą.
8. Napisz program przechowujący w statycznej tablicy elementy struktury osoba (imię, nazwisko, wiek) oraz funkcje dodające i usuwające elementy takiej tablicy (zastanów się, co w przypadku tablicy statycznej znaczy „dodać element”, jak oznaczać elementy usuwane, a może można sobie poradzić z ich usuwaniem bez konieczności oznaczania). Przekształcić powyższy program tak, aby elementy tablicy były wskaźnikami. W funkcjach dodawania i usuwania elementów zastosuj funkcje rezerwujące i zwalniające pamięć.
9. Z programu zliczającego przecięcia zera wydziel funkcję sprawdzającą czy nastąpiło przecięcie zera, umieść ją w oddzielnym module i skompiluj całość.