WYDZIAŁ FIZYKI
UNIWERSYTET WARSZAWSKI

# Programowanie zaawansowane FM i NI

## Wykład 5

# Tablice, napisy, alokacja dynamiczna

Krzysztof Piasecki

Semestr letni roku akad. 2023-24

krzysztof.piasecki@fuw.edu.pl

- **Tablica (array)** – to uporządkowany zbiór danych tego samego typu, w pamięci jako ciągły blok.
  Ilość elementów tablicy = **rozmiar (size)**.

  Tu omówimy **tablice statyczne**. „Statyczne” oznacza tu: na etapie kompilacji rozmiar jest ustalony.

  *Uwaga*: w praktyce na PC rozmiar pamięci na tablice statyczne (i inne lokalne zmienne) wynosi ~ 8 MB.
  Ten obszar pamięci nazywa się **stos** (**stack**).

- Deklaracja **tablicy** 5-elementowej i przypisanie **zbioru** 5-elementowego:

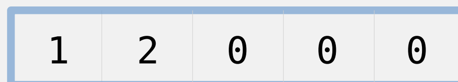
```
int tab[5] = { 8, 3, 6, 5, 7 } ;
```

Poniżej, kompilator zadeklaruje taki rozmiar, jaki wynika ze zbioru.

```
int tab2[] = { 1, 2, 3, 4, 5 } ;
```

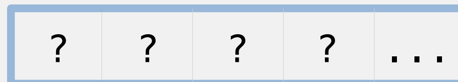
Poniżej: rozmiar będzie 5, ale inicjujemy 2 pierwsze elementy. Reszta = 0.

```
int tab3[5] = { 1, 2 } ;
```




Poniżej: deklarując, można nie przypisywać (*niezbyt bezpieczne*)

```
int tab4[100] ;
```




Poniżej: do wymiarowania można użyć „stałej zmiennej”.

```
const int Dim = 20;
int tab5 [Dim] ;
```

## Komunikacja z tablicą statyczną

- Wpisywanie do danego elementu:

```
tab[2] = -3 ;
```

Dlaczego komputer wstawił w pozycję trzecią z lewej, a nie drugą ?
→ w C/C++ jest umowa, że **elementy indeksujemy od 0**.

- Czytanie elementu:

```
cout << tab[2] ;                                → -3
```

Rozmiar tablicy w bajtach:
... i w liczbie elementów:

```
cout << sizeof (tab)                            → 20
     << sizeof (tab) / sizeof(tab[0])           → 5
```

*Uwaga*: te operacje, są *niestety* dozwolone:

```
tab[-1] = 5;    tab[ sizeof(tab) + 15 ] = 123;
```

Komputer będzie sięgał do sąsiednich komórek pamięci i nie wypisze błędu (*chyba, że będzie to obszar poza stosem*) Nie powinniśmy do tego dopuszczać.

- Pętle po elementach robimy albo. wg wcześniej poznanych sposobów,

```
for (int i = 0; i < sizeof (tab) / sizeof (tab[0]); i++) {
  cout << tab[i] << ' ';
}
```

albo przez **pętlę zakresową**:

```
for ( int& elem : tab ) {
  elem++ ;
  cout << elem ;
}
```

w każdym kroku `elem` = kolejny element tablicy.

## Przekaz tablicy statycznej do funkcji

[Link]

Funkcja przyjmująca tablicę statyczną, w argumentach wejścia ma postać:

① `typ nazwa[rozmiar]` , albo

② `typ nazwa[]` , albo

③ `typ* nazwa` .

W przypadkach ① i ② typ obiektu i tak sprowadzi się do: `typ*` .

- W miejscu wywołania podajemy nazwę tablicy. Nieprzypadkowo: nazwa tablicy ... to wskaźnik na jej zerowy element!

- Wywołana funkcja nigdy nie pozna rozmiaru tablicy statycznej 😔 . Chcąc przekazać rozmiar, robimy to przez osobny argument wejścia.

```
F1:  5
F1:  8  2

F2:  8  3  6  5  7
```

```cpp
#include<iostream>
using namespace std;

void funtab1 ( int T[] ) {
   cout << "funtab1: " << T[3]  << endl;
   cout << "funtab1: " << sizeof(T) << ' '
        << sizeof(T) / sizeof(T[0]) << "\n\n";
}

void funtab2 ( int T[] , int size ) {
   cout << "funtab2: ";
   for (int i = 0 ; i < size ; i++)
      cout << T[i] << ' ';

   cout << endl;
}

int main () {
   int tab[5] = { 8 ,3 ,6 ,5 ,7 };

   funtab1 ( tab );
   funtab2 ( tab, sizeof(tab)/sizeof(tab[0]) );
}
```

- ## Tablice a wskaźniki

[Link]

Adres początku tablicy to:
albo `&tab[0]` , albo `tab`

Adres i-tego elementu tablicy to:
albo `&tab[i]` , albo `tab+i`

Wartość i-tego elementu tablicy to:
albo `tab[i]` , albo `*(tab+i)`

Pętlę po elementach tablicy
można zamienić
na pętlę po kolejnych adresach.
Indeksem musi stać się wskaźnik.
Iterujemy przez + , - , ++ , --

Tak wyciągamy nr elementu

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5   int tab[3] = { 8 , 3 , 6 };
6
7   cout << &tab[0] << ' ' << tab << "\n\n";
8
9   for ( int i = 0 ; i < 3 ; i++ ) {
10      cout << i
11          << ' ' << &tab[i] << ' ' <<   tab+i
12          << ' ' <<  tab[i] << ' ' << *(tab+i)
13          << endl;
14  }
15  cout << endl;
16
17  for ( int* p = tab ; p < tab + 3 ; p++ ) {
18      cout << p - tab << ' ' << p << ' '
19          << *p << endl;
20  }
21 }
```

- **Określanie typu: *„reguła prawo-lewo”* (*right-left rule*) .** ( *dawniej: reguła spiralna* )

Określając typ zmiennej, kompilator działa według konkretnego schematu.

① Zaczynamy od nazwy własnej. Od razu – wyjątek:
Jeśli jest objęta nawiasami i poprzedzona & lub *, to obiekt jest referencją lub wskaźnikiem.

② Idziemy w prawo. Jeśli jest tam [x] – obiekt jest tablicą o rozmiarze x.
(chyba że np. [x][y] – obiekt jest tablicą dwuwymiarową (itd.)
Jeśli jest ( *zmienne* ) – obiekt jest funkcją, ew. z argumentami wejścia.

③ odtąd kręcimy się zgodnie ze wskazówkami zegara, de-facto coraz bardziej na lewo.

Przykład od autora reguły:

```
         +---------+
         |  +---+  |
         |  ^   |  |
    char* tab [10] ;
     ^   ^      |  |
     |   +------+  |
     +-------------+
```

tab jest tablicą 10 elementów.
... której każdym elementem jest wskaźnik
... na zmienną typu `char`.

```
int (& ABC) [10][20] = ... ;
```

ABC jest referencją
... na tablicę wymiaru 10x20
... której każdy element ma typ int

```
unsigned int** MyPtr2 ;
```

`MyPtr2` jest wskaźnikiem
... do trzymania adresu wskaźnika,
... do adresu zmiennej typu `unsigned int`

- **C-string**: napis wykonany jako statyczna tablica char'ów (tzw. napis „*w stylu C*")

Typowa deklaracja C-stringu :

```
char Text[] = "Moj napis" ;
```

*Uwaga*: zaraz po napisie komputer zawsze dostawi **znak końca**. Jest to **char o wartości 0**.

→ Dzięki temu, funkcje działające na C-stringach rozpoznają koniec napisu. Np. tak działa `cout << napis`

→ długość tablicy = liczba znaków + 1. Ale funkcja **`strlen`** (podaje długość napisu) pominie znak końca.

→ gdy przekazujemy C-string do funkcji, nie musimy osobno podawać rozmiaru.

N-ta litera napisu to `Text[N-1]` .

Do porównywania napisów służy **`strcmp`** . Jeśli napisy są identyczne, `strcmp` zwróci 0 .

- [Link] do biblioteki `cstring`. M.in. **`strcpy`**.

[Link]

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  void funNapis ( char Text[] ) {
6    cout << "[funNapis]: " ;
7    for (int i = 0 ; Text[i] != 0 ; i++ )
8      cout << Text[i] ;
9  }
10
11 int main () {
12   char napis[] = "Kodowanie w C++" ;
13   cout << "Napis [" << napis << "] ma "
14        << strlen (napis) << " znaków. \n"
15        << "Piaty znak to: " << napis[4] << endl;
16
17   if (napis[4] == 'w') cout << "Jest to w.\n";
18   else                 cout << "To nie  w.\n";
19
20   char napis2[] = "Kodowanie w C--" ;
21   if ( strcmp (napis, napis2) == 0 )
22     cout << "Napis to: " << napis2 <<  endl;
23   else
24     cout << "Napis inny niz: " << napis2 << endl;
25
26   funNapis (napis) ;
27 }
```

- **Wywoływanie programu z opcjami** (argumenty wywołania funkcji `main`)

- Możemy do programu przyjmować opcje z linii wywołania (tzw. **argumenty wywołania**). Funkcja `main` powinna mieć nagłówek:

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

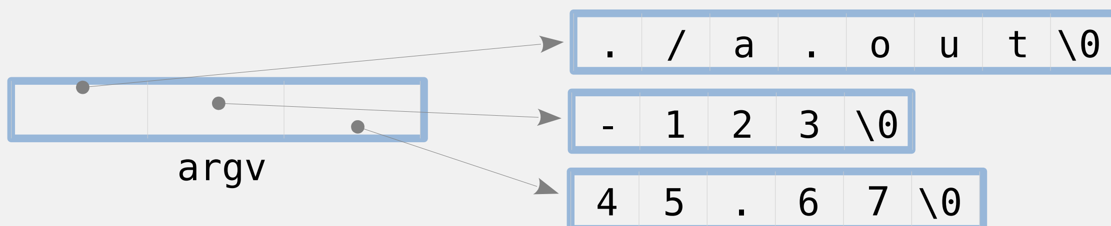
Liczba opcji [+1], którą użytkownik wpisał w danym wywołaniu. [+1] , bo wpisał też nazwę aplikacji.

Tablica z adresami, gdzie zaczynają się opcje w formie C-stringów

- Przypuśćmy, że użytkownik wpisał:

```
$ ./a.out -123 45.67
```

wtedy `argc = 3` , a `argv` ma postać:




[Link]

```
1  #include <iostream>
2  #include <cstring>
3  #include <cstdlib>
4  using namespace std;
5
6  int main ( int argc,  char* argv[] )
7  {
8    for (int i = 0 ; i < argc ; i++ ) {
9      cout << argv[i] << endl;
10   }
11   int    ivalue = atoi (argv[1]) ;
12   double dvalue = atof (argv[2]) ;
13
14   cout << "Rozczytane argumenty: "
15        << ivalue <<'\t'<< dvalue << endl;
16
17   if ( strcmp (argv[0], "./a.out") == 0 ) {
18     cout << "Program wykonany przez ./a.out";
19   }
20 }
```

- Potrzebna jest jeszcze konwersja C-stringu do `int` lub `double`. Są to funkcje **`atoi`** i **`atof`** w bibliotece `cstdlib`.

- **Dynamiczna alokacja pamięci (dynamic memory allocation)**

to odmienny sposób na tworzenie obiektów.

| Cecha | Statyczna alokacja | Dynamiczna alokacja |
|---|---|---|
| Rozmiar tablicy | zafiksowany w kodzie | można uzależnić od zmiennej |
| Zakres zmiennej | wnętrze $\{\ \}$ | przeżywa poza $\{\ \}$ |
| Limit pamięci | typowo $\sim$ 8 MB | cała dostępna pamięć |
| Rozmiar tablicy | wydobywalny (sizeof) | niewydobywalny |
| Sposób skasowania | komputer sam kasuje | programista kasuje |
| Organizacja pamięci | stos (stack) | sterta (heap) |

Zwykle, **sterta** (**heap**) jest obszarem pamięci znacznie pojemniejszym,
zazwyczaj ograniczona limitem pamięci przydzielonej przez system.

- **Dynamiczna alokacja pamięci (dynamic memory allocation)**

⊙ **Tworzenie obiektu**: operator **new**

Podajemy mu typ obiektu do utworzenia.
new robi wpis do **tablicy alokacji**
i zwraca nam adres utworzonego obiektu.
Ten adres chowamy do wskaźnika:

```
int* Iptr = new int ;
```

Jeśli chcemy utworzyć tablicę obiektów, dokładamy [] .
new zwróci adres *początkowego* elementu tablicy.
Ale w tablicy alokacji wiadomo, że to tablica.

```
int* Tptr = new int[5] ;
```

⊙ **Kasowanie**: operator **delete** (lub **delete[]** do tablic)
Podajemy mu wskaźnik, który trzyma adres obiektu.
delete usuwa wpis z tablicy alokacji.

```
delete   Iptr ;
delete[] Tptr ;
```

Uwaga: delete nie zeruje wskaźnika.
Jeśli chcemy go wyzerować, przypisujemy mu **nullptr**.

[Link]

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5    int* Iptr = new int ;
6    *Iptr = 123;
7    cout << "Iptr: "    << Iptr
8         << " Value:  "<< *Iptr << endl;
9
10   int* Tptr = new int [5] ;
11   Tptr[3] = -456;
12   cout << "Tptr: "    << Tptr
13        <<" Tptr[3] " << Tptr[3] ;
14
15   int Dim;
16   cout << "\nPodaj rozmiar fTab: ";
17   cin >> Dim;
18   float* fTab = new float [Dim] ;
19   cout << "ftab: " << fTab << ", size? "
20        << sizeof(fTab) / sizeof (fTab[0]);
21
22   delete   Iptr ;
23   delete[] fTab ;
24                    cout << endl << Iptr ;
25   Iptr = nullptr; cout << endl << Iptr ;
26 }
```

- `string`: napis w stylu C++ .
  - ☞ więcej bezpieczeństwa, np.:

    `at(i)` : odczyt/zapis i-tego znaku,
    ale po sprawdzeniu, czy i ∈ [ 0 , `size()` )
  - ☞ więcej narzędzi do manipulacji
  - ☞ możliwość sekwencji poleceń
  - ☞ przechodzi do funkcji jako 1 obiekt.

⊙ [Link] do poleceń z biblioteki `string`. Np:

| | |
|---|---|
| `size ()` | długość |
| `at (i)` | litera na pozycji i |
| `to_string (..)` | konwersja liczby na string |
| `find ('a')` | pozycja wystąpienia 'a' |
| `str1 + str2` | połącz dwa napisy |
| `compare (s1, s2)` | porównaj dwa stringi |
| `erase (p,n)` | od pozycji p skasuj n liter |
| `insert (p,"X")` | w pozycję p wstaw X |
| `replace (p,n,"X")` | wymień n liter od poz. p → na X |
| `substr (p,n)` | utwórz string z poz. [p, p+n) |

[Link]

```
1  #include <string>
2  #include <iostream>
3  #include <typeinfo>
4  using namespace std;
5
6  int main () {
7   float cena = 4.5 ;
8   string S1 = "Cena napoju: " ,
9          S2 = S1 + to_string( cena ) + " zl.";
10  cout << S2 << "\nSize: " << S2.size() ;
11  cout <<'\n'<< S2[0] <<' '<< S2.at(0) << endl;
12
13  string S3 = S2.substr (S2.find('.') -1 , 4);
14  S2 = S1 + S3 + " zl.";
15  cout << S3 << endl << S2 << endl ;
16
17  S2.replace ( 18 , 2 , "EUR" )
18    .erase   (  0 , 1 )
19    .insert  (  0 , "Nowa c") ;
20  cout << S2 << endl;
21
22  string Snum1 = "-123" , Snum2 = "-4.567" ;
23  int    num1 = stoi ( Snum1 ) ;
24  double num2 = stod ( Snum2 ) ;
25  cout << num1 << ' ' << num2 ;
26 }
```

- **Konwersje `C-string` ↔ `string`**

Przejście `C-string` → `string` jest proste.
Gdy mamy `char CS[] = "abc"` , to

```
string S = string (CS)
```

Ale przejście `string` → `C-string`
w sposób prosty:

```
const char* CS = S.c_str()
```

daje połowiczny sukces, bo:

① CS jest const,
czyli nie wolno go zmienić

② CS dotyczy tego samego adresu, co S,
więc dowolna zmiana S zmienia i CS.

Najprostszym obejściem jest
skopiowanie z CS komendą
`strcpy` z biblioteki `cstring`.

[Link]

```
1  #include <string>
2  #include <cstring>
3  #include <iostream>
4  using namespace std;
5
6  int main () {
7   char cText[] = "Programowanie C++" ;
8   string sText = string ( cText ) ;
9   cout << "c-string 1: " << cText << endl;
10  cout << "  string 1: " << sText << "\n\n";
11
12  const char* cTextOrig = sText.c_str () ;
13  cout << "c-string 2: " << cTextOrig << endl;
14
15  sText.at(0) = 'X';
16  cout << "c-string 2: " << cTextOrig << "\n\n";
17  sText.at(0) = 'P';
18
19  char cTextCopy [ sText.size() + 1 ] ;
20  strcpy ( cTextCopy , sText.c_str() ) ;
21  sText.at(0) = 'Q';
22  cout << "  string 1: " << sText     << endl;
23  cout << "c-string 3: " << cTextCopy << endl;
24 }
```

- **Wyszukiwanie w stringu**. Niech `string S = "Tak trzeba zyc!" ;`

  Pierwsze wystąpienie 'a' `S.find ('a')` → 1

  Poszukajmy 'a' począwszy od poz. 5: `S.find ('a', 5)` → 9

  Gdy znak się nie znalazł: `S.find ('x')` → zwróci `string::npos`

  Gdy szukamy całego słowa: `S.find ("zyc")` → 11

- To nie koniec. Z **listy kandydatów** poszukajmy,

  gdy pierwszy raz coś wystąpi:

  `S.find_first_of ("abc")` → 1

  gdy pierwszy raz wystąpi coś spoza listy:

  `S.find_first_not_of ("Tt")` → 1

  gdy ostatni raz coś wystąpi:

  `S.find_last_of ("xyz")` → 12

  gdy ostatni raz wystąpi coś spoza listy:

  `S.find_last_not_of ("abc")` → 14

[Link]

```
#include <iostream>
#include <string>
using namespace std;

int main () {
 string S ("Tutaj znikna samogloski.");
 size_t found = S.find_first_of ("aeiouy");

 while ( found != string::npos )  {
  S.at ( found ) = '_' ;
  found = S.find_first_of ("aeiou", found+1);
 }
 cout << S << '\n';
}
```