Programowanie obiektowe

Dynamiczne struktury danych, zarządzanie pamięcią

Stos a sterta

- Zalety używania dynamicznego przydzielania pamięci:
  - Może ona być dzielona między różnymi obiektami w programie.
  - Jej rozmiar może być ustalany podczas działania programu.

```
int a = 2;
```

STOS

STERTA

a 2

```
int* a = new int;
```

STOS

STERTA

a → ? *a

Wskaźniki w C++

- Zmienna wskaźnikowa – zmienna, która przechowuje adres innego bytu programistycznego (obiektu, zmiennej) w pamięci.
- Wskaźnik musi mieć podany typ, na jaki wskazuje.

```
1  int jakasLiczba = 10;
2  int* jakisWskaznik;
3
4  jakisWskaznik = &jakasLiczba;
5  cout << *jakisWskaznik;
```

Wskaźniki w C++

```
1  int jakasLiczba = 10;
2  int* jakisWskaznik;
3
4  jakisWskaznik = &jakasLiczba;
5  cout << *jakisWskaznik;
```

Ad1: deklaracja zmiennej typu całkowitoliczbowego
Ad2: deklaracja **wskaźnika na int**
Ad4: **przypisanie** do wskaźnika **adresu** zmiennej
Ad5: **wyłuskanie** wartości na którą wskazuje wskaźnik

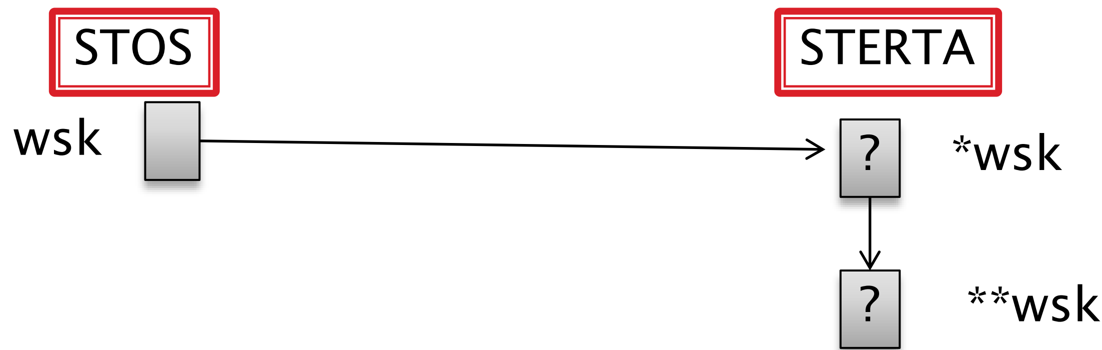
Wskaźniki w C++

```cpp
#include <iostream>

using namespace std;

int main() {

int jakasLiczba = 10;
int* jakisWskaznik;

jakisWskaznik = &jakasLiczba;
cout << "Adres wskazanika " << jakisWskaznik << endl;
cout << "Adres zmiennej " << &jakasLiczba << endl;
cout << "Wartosc " << *jakisWskaznik << endl;

return 0;
}
```

```
Adres wskazanika 0x61fe14
Adres zmiennej 0x61fe14
Wartosc 10
```

Alokacja i dealokacja

```
int** wsk;
wsk = new int*;
*wsk = new int;
```




```
new int; // Wyciek pamięci
```

```
int* liczba = new int;
delete liczba;
```

Do każdego **new** powinno być komplementarne **delete**.

Unikanie pułapek – mechanizm wyjątków

Wersja zakładająca, że operator *new* generuje wyjątek w przypadku braku wolnej pamięci:

```
char * s = 0;
int n;

// Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n

try
{
  s = new char [ n ];
  Kod do obsługi napisów
  . . .
  delete [] s;
}
catch( ... ) // Wersja uproszczona, dokładniej catch( std::bad_alloc & e )
{
  cout << "Brak pamięci dla wykonania tej operacji";
}
```

Tablice – typy proste

- Podobne do Javy:

typ nazwa[rozmiar]

Tego typu tablice tworzone są na stosie.

```
21  typ nazwa_tablicy[rozmiar];
22
23  int tablica[10];
24
25  int tablica[4] = {2,4,6,8};
26
27  int tablica[100] = {1,};
28
29  int tablica[] = {1, 2, 3, 4, 5, 6, 7, 10};
30
```

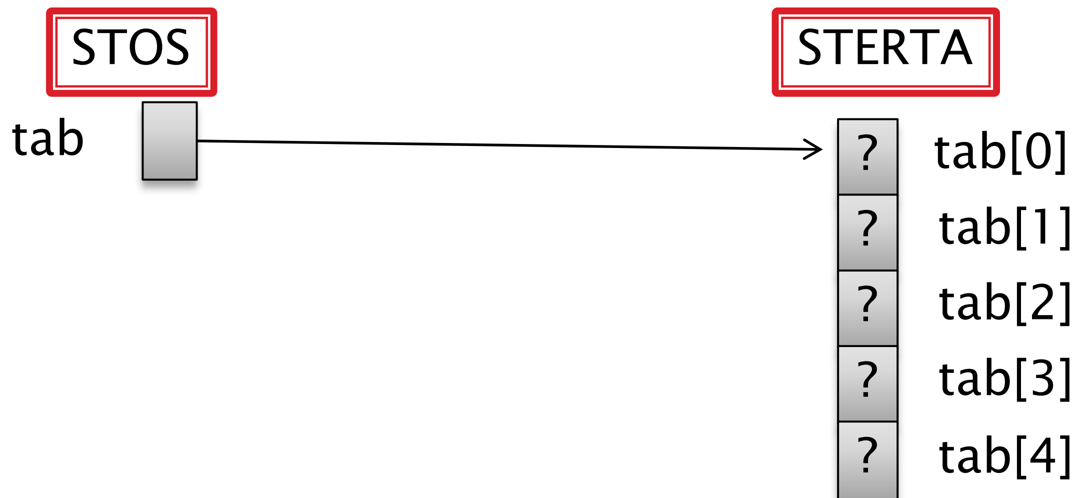
```
23  int main(int argc, char *argv[])
24  {
25      int tablica[5] = {1,2,3,4,5};
26      for (int i=0; i<(sizeof tablica / sizeof *tablica); ++i) cout << tablica[i] <<endl;
27      system("PAUSE");
28      return 0;
29  }
```

Przekazywanie „tablic” do funkcji

```
7  void test(int tab[]) {
8          for(int i=0;i<10;++i) tab[i]+=1;
9          }
10
11 int main(int argc, char *argv[])
12 {
13 int tab[10]={1,1,1,1,1,1,1,1,1,1};
14 cout << tab[0] << endl;
15 test(tab);
16 cout << tab[0] << endl;
17
18 system("PAUSE");
19 return 0;
```

Tablice alokowane dynamicznie

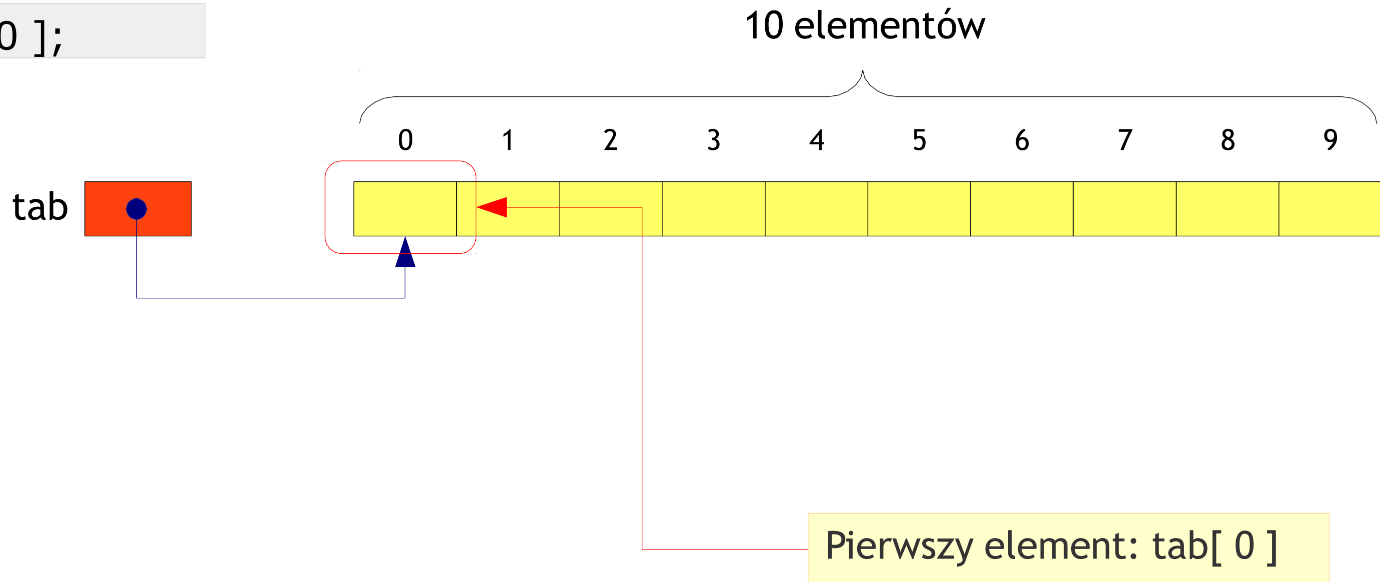
```
int* tab = new int[5];
delete[] tab;
```



Tablice dynamiczne != Tablice alokowane dynamicznie

Nazwa tablicy jako wskaźnik na jej początek

Nazwa tablicy jest interpretowana jako *ustalony wskaźnik* na jej początek (pierwszy element).

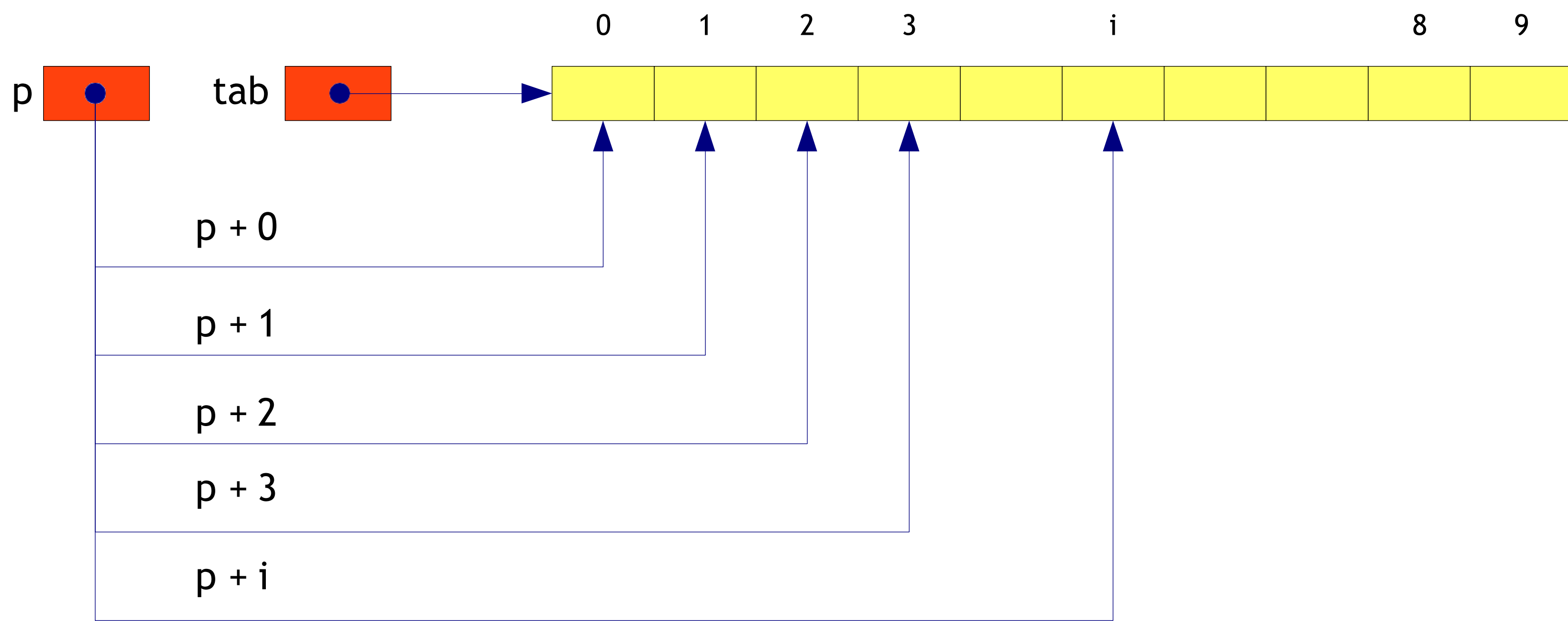


Nazwa tablicy jako wskaźnik na jej początek, cd. ...

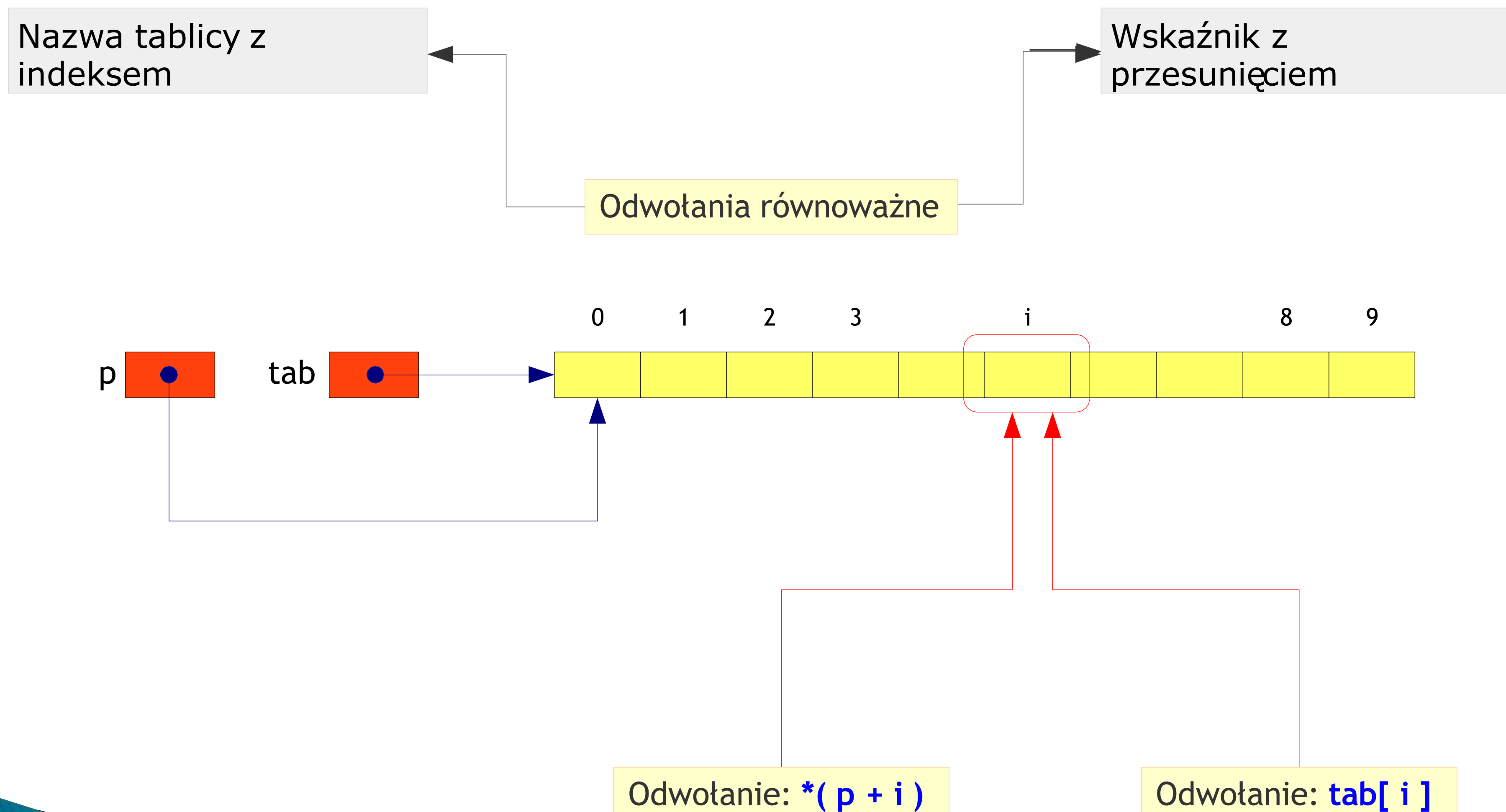
```
tab[ 0   ]  = 5;
tab[ 1   ]  = 1;
tab[ 2   ]  = 10;
. . .
tab[ i ] = 22;
```

Odwołania równoważne

```
*p = 5
*( p +   1  )  =  1
*( p +   2  )  =  10
. . .
*( p + i ) = 22
```



Nazwa tablicy jako wskaźnik na jej początek, cd. ...



Nazwa tablicy to *ustalony* wskaźnik na jej początek

Nazwa tablicy jest *ustalonym* (*niemodyfikowalnym*) wskaźnikiem na pierwszy jej element. Nazw tablic *nie wolno modyfikować*! Zwykłe wskaźniki *można*.

```
int tab[ 10 ];
int * p = tab;

tab = p;
tab++;
```

Źle

```
p = tab + 8;
p++;
```

OK

Wskaźniki a tablice

- W C++ można stosować arytmetykę wskaźników (do wskaźnika dodawać i odejmować liczby całkowite). Kompilator sam pomnoży dodawaną liczbę przez rozmiar typu wskaźnika aby dodać do adresu odpowiednią liczbę bajtów.

```
1  int tab[4];     //tab jest typu int *
2  char tab2[4][7]; //tab2 jest typu char **
3  //jest to podwójny wskaźnik bo tablica jest dwuwymiarowa
4
5  *tab = 0;          //równoznaczne z tab[0]=0
6  *(tab+1) = 5;      //równoznaczne z tab[1]=5
7  **tab2 = 'a';          //równoznaczne z tab2[0][0]='a'
8  *(tab2[4]+5) = 'c';  //równoznaczne z tab2[4][5]='c'
```

Tablice a wskaźniki c.d.

- Odwoływanie się do tablicy jednowymiarowej:

```
int tab[6];
int* wskTab = tab;
wskTab[4] = 1;
```

- Przekazywanie do funkcji:

```
void Przetwarzaj(int* tablica) { // (int tablica[])
  for (int i = 0; i < 5; i++) {
      tablica[i] = -5;
  }
}
int* tab_dyn = new int[2];
tab[0] = 2;
tab[1] = 3;
int tab_stos[] = {2, 3};
Przetwarzaj(tab_dyn);
Przetwarzaj(tab_stos);
Przetwarzaj(&tab_stos[0]);
```

Tablice wielowymiarowe

```
char statki[3][3];
statki[1][1] = 'X';
```

STOS

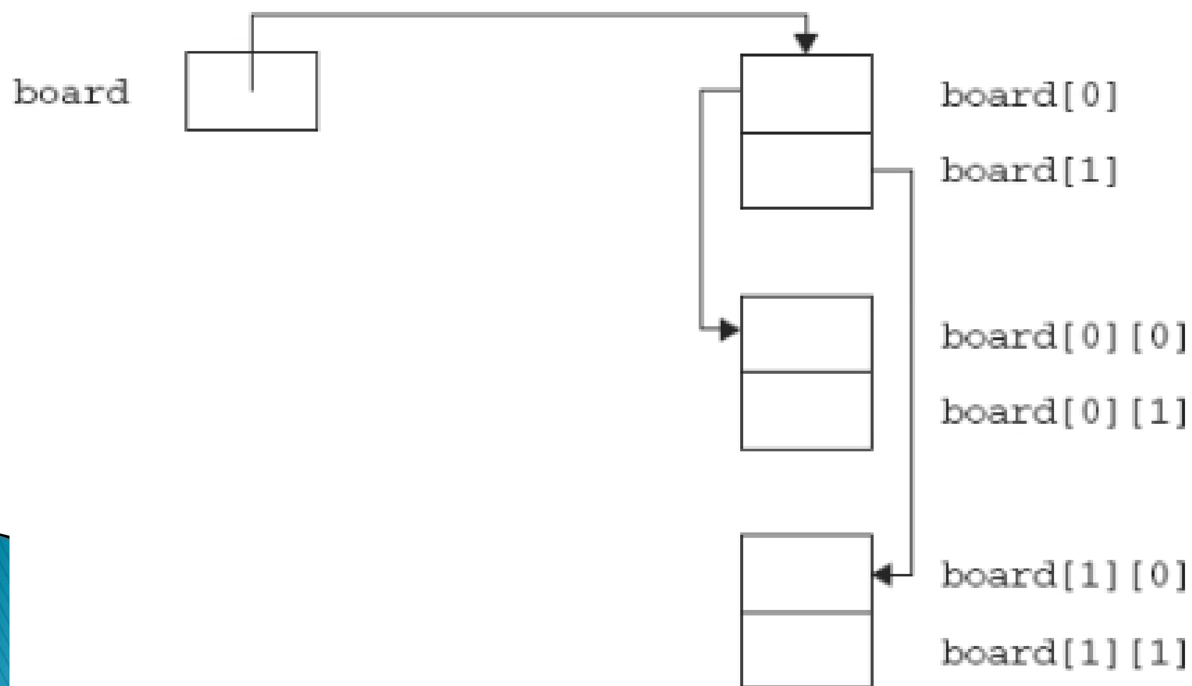
?	tab[0][0]
?	tab[0][1]
?	tab[0][2]
?	tab[1][0]
?	tab[1][1]
?	tab[1][2]
?	tab[2][0]
?	tab[2][1]
?	tab[2][2]

Wielowymiarowa tablica alokowana dynamicznie

- Czy tak można? Nie.

```
const int i = 2, j = 2;
char** board = new char[i][j];
```

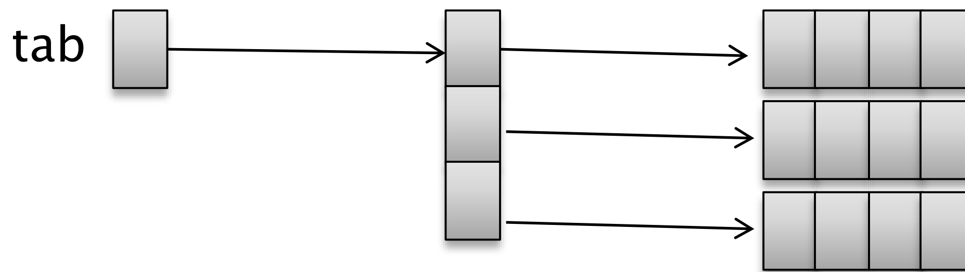
STOS STERTA



Dwuwymiarowa tablica alokowana dynamicznie

- Prawidłowa konstrukcja (3 x 4)

```
char** tab = new char*[3];
for (size_t i = 0; i < 3; i++)
 tab[i] = new char[4];
```




```
for (size_t i = 0; i < 3; i++)
 delete[] tab[i];
delete[] tab;
```

Zwiększamy wymiar

```
#include <iostream>

using namespace std;

void suma(int tab[][2][2]) {
    tab[1][1][1] = 5;
    cout << tab[1][1][1] << endl;
}

int main()
{
    int tab[2][2][2] = {{{1,2},{3,4}},{{5,6},{7,8}}};
    suma(tab);
    return 0;
}

```

Ćwiczenia 1

- Stwórz klasę `Fibb`, której głównym celem będzie stworzenie tablicy alokowanej na stercie i wypełnienie jej zgodnie z regułami ciągu Fibonacciego (pierwszy i drugi element = 1, każdy następny to suma dwóch poprzednich: 1,1,2,3,5,8, itd.).
  Klasa powinna posiadać **pole** prywatne `rozmiar`, definiujące wielkość (liczbę elementów) tablicy, publiczny **konstruktor ogólny** definiujący tablice o określonym rozmiarze oraz **metodę** `generujCiag()`, której zadaniem jest uzupełnienie stworzonej tablicy (zgodnie z regułami ciągu Fibonacciego) jak również jej wyświetlenie na ekranie.

Ćwiczenia 2

- Korzystając z następującej struktury (symbolizującej element stosu):

```
struct elem {
    int dane;
    elem * nast;
}
```

- Zaimplementuj podstawowe operacje stosowe (w ramach klasy `Stack`):
  - Położenie elementu `x` na wierzchołku stosu.

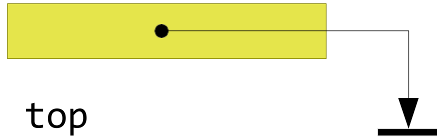
  void push(int x)
  - Usunięcie ostatnio odłożonego elementu i zwrócenie go jako wartości funkcji.

  int pop()
  - Zwrócenie elementu znajdującego się na wierzchołku stosu bez jego usuwania.

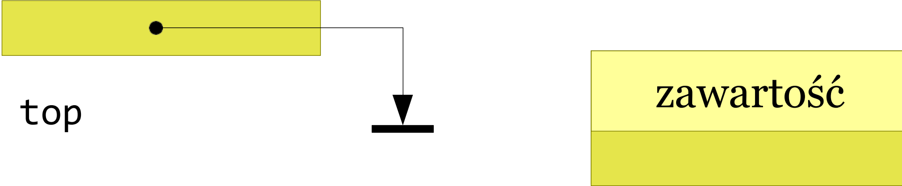
  int topEl()
  - Sprawdzenie czy stos jest pusty.

  bool empty()

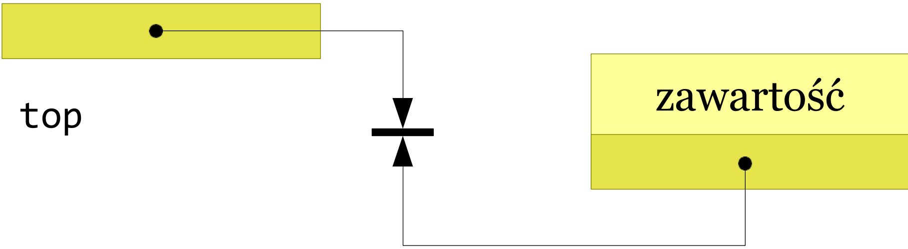
Stos jako lista – pusty stos



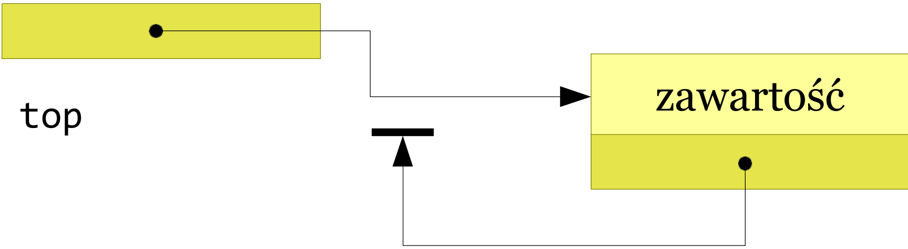
Stos jako lista – dodanie elementu, etap pierwszy



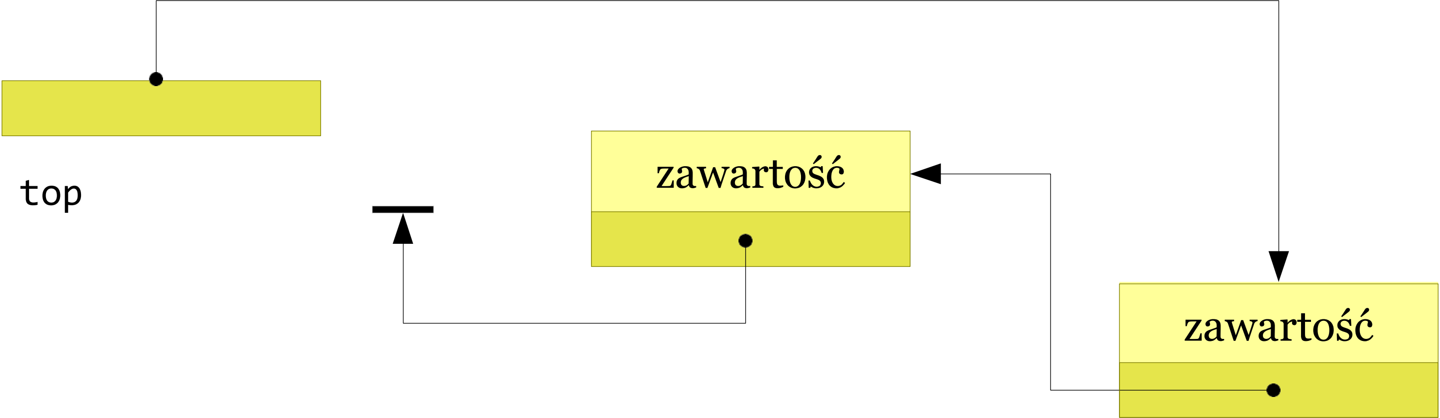
Stos jako lista – dodanie elementu, etap drugi



Stos jako lista – dodanie elementu, etap trzeci



Stos jako lista – kolejny element



Ćwiczenia 3

- Zrealizować obiektowo grę w statki na planszy generowanej dynamicznie (od 5x5 do 10x10). Plansza powinna być realizowana w formie tablicy dwuwymiarowej alokowanej na stercie. Raz strzela użytkownik, raz komputer.
- Dozwolone okręty:

  4 x jednomasztowce

  Zaprogramować sytuacje:
  - „Trafiony – zatopiony”,
  - „Pudło”,
  - „Strzelaj w plansze”.

Ćwiczenia 4

- Korzystając z następującej struktury (symbolizującej element listy jednokierunkowej):

```
struct elem {
    int dane;
    elem * nast;
}
```

- Zaimplementuj podstawowe operacje stosowe (w ramach klasy `List`):
  - Dodanie elementu x na początek listy.

  void push_front(int x)
  - Dodanie elementu x na koniec listy.

  void push_back(int x)
  - Usunięcie elementu znajdującego się na początku listy i jego zwrócenie.

  int pop_front()
  - Wypisanie na ekranie wszystkich elementów listy.

  void printl()