Programowanie obiektowe C++

Programowanie zorientowane obiektowo

Wykład 4

Witold Dyrka
witold.dyrka@pwr.wroc.pl

29/10/2012

Prawa autorskie itp.

Dzisiejszy wykładu powstał głównie
w oparciu o slajdy Bjarne Stroustrupa
do kursu Foundations of Engineering II (C++)
prowadzonego w Texas A&M University
http://www.stroustrup.com/Programming

Program wykładów

1. Wprowadzenie. Obsługa błędów. Klasy (8/10)
2. Abstrakcja i enkapsulacja: klasy i struktury. (15/10)
3. Polimorfizm: przeładowanie funkcji i operatorów. Konwersje (22/10)
4. **Zarządzanie pamięcią: konstruktor, destruktor, przypisanie (29/10)**
5. Dziedziczenie. Polimorfizm dynamiczny (5/11)
6. Programowanie uogólnione: szablony (12/11)

Materiały

Literatura

- Bjarne Stroustrup. *Programowanie: Teoria i praktyka z wykorzystaniem C++*. Helion (2010)
- Jerzy Grębosz. *Symfonia C++ standard.* Edition 2000 (2008)
- Dowolny podręcznik programowania zorientowanego obiektowo w języku C++ w standardzie ISO 98

Plan na dziś

- **Zarządzanie pamięcią w klasie** na przykładzie **vector**-a
  - Pamięć wolna i jej wyciekanie
  - Kopiowanie obiektów
    - Konstruktor kopiujący
    - Operator przypisania

  - *Zarządzanie zasobami przez zakres*
  - *Gwarancje bezpieczeństwa wyjątków*

Przyjrzymy się implementacji **vector**-a

- **vector** jest najbardziej użytecznym kontenerem STL
  - prosty
  - zwarty sposób przechowywania elementów danego typu
  - efektywny dostęp do elementów
  - przechowywanie dowolnej (zmiennej) liczby elementów
  - opcjonalnie sprawdzanie zakresu

- **vector** jest domyślnym kontenerem języka C++
  - stosuj go zawsze (np. do przechowywania tablicy)
    - chyba, że masz dobry powód by wybrać coś innego

vector a pisanie kodu od podstaw

- Sprzęt zapewnia pamięć i adresowanie
  - tzw. niski poziom
  - brak typów danych, fragmenty pamięci o stałym rozmiarze
  - żadnego sprawdzania, ale tak szybko jak architektura pozwala
- Twórca aplikacji potrzebuje czegoś takiego jak **vector**
  - operacje wysokiego poziomu
  - kontrola typów i zmienny rozmiar pamięci
  - sprawdzanie zakresu w czasie działania
  - prawie optymalna szybkość

vector a pisanie kodu od podstaw (2)

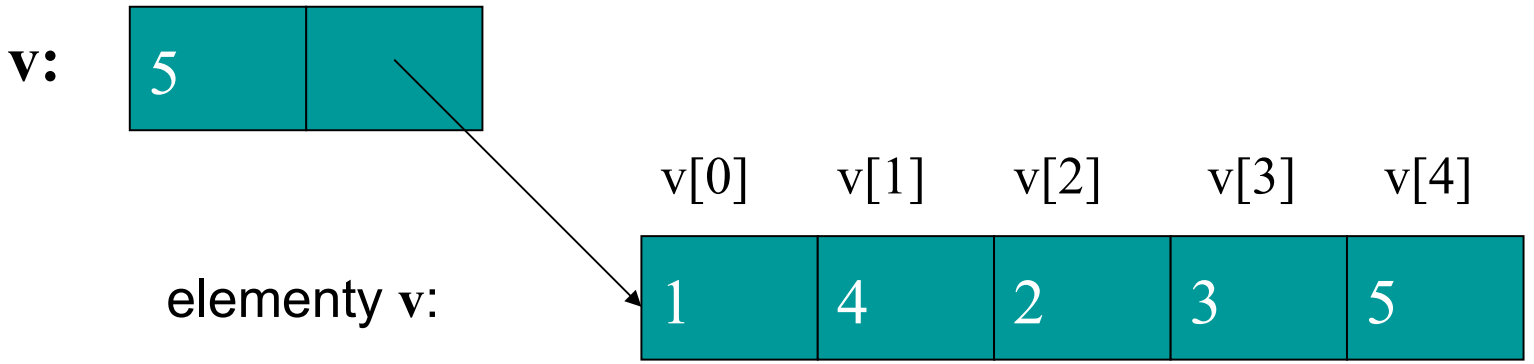
- Tam, blisko sprzętu, życie jest proste i brutalne
  - musisz zaprogramować wszystko sam
  - nie pomoże Ci kontrola typów
  - błędy w czasie działania poznaje się po tym, że dane uległy uszkodzeniu lub program się wysypał
- O nie, chcemy się wyrwać stamtąd tak szybko jak się da
  - chcemy być produktywni i pisać niezawodne programy
  - chcemy używać języka odpowiedniego dla ludzi
- Dlatego używamy wektorów itp.
  - ale chcemy też poznać techniki tworzenia nowych klas służących do pracy ze strukturami danych

Po co przyglądać się implementacji **vector**-a ?

- Ilustracja kilku ważnych koncepcji:
  - Wolna pamięć (sterta)
  - Kopiowanie obiektów
  - Dynamicznie rosnące struktury danych
- Demonstracja technik:
  - bezpośredniego zarządzania pamięcią
  - porządnego projektowania i kodowania klas

Projektujemy typ **vector** (1)

- Założenia
  - reprezentuje pojęcie tablicy jednowymiarowej
  - może przechowywać dowolną i zmienną liczbę elementów typu **double**
  - zapewnia inicjowanie elementów oraz dostęp do nich



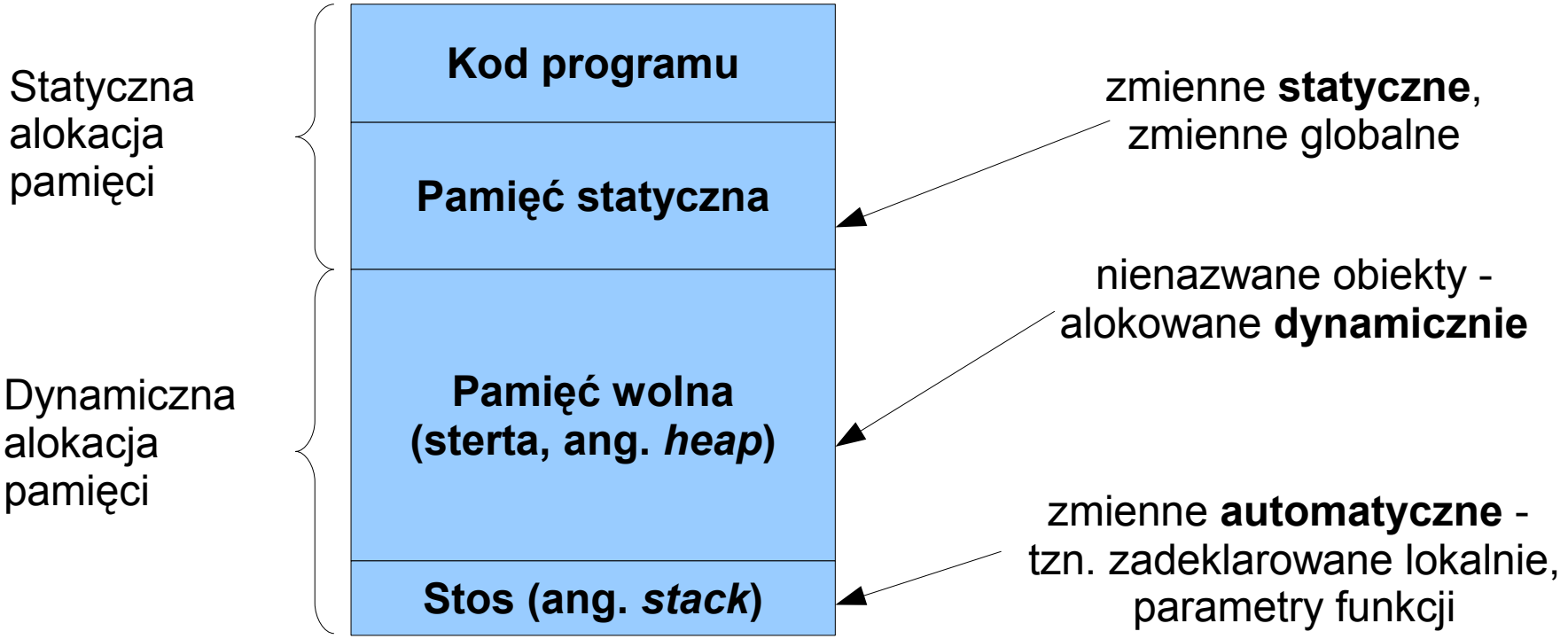
Projektujemy typ **vector** (2)

- Założenia
  - reprezentuje pojęcie tablicy jednowymiarowej
  - może przechowywać dowolną i zmienną liczbę elementów typu **double**
  - zapewnia inicjowanie elementów oraz dostęp do nich

- *Gdzie przechowywać tablicę dowolnego, tzn. nieustalonego w czasie kompilacji, rozmiaru?*
  - *w pamięci wolnej*

Pamięć komputera

- tak jak widzi ją program:

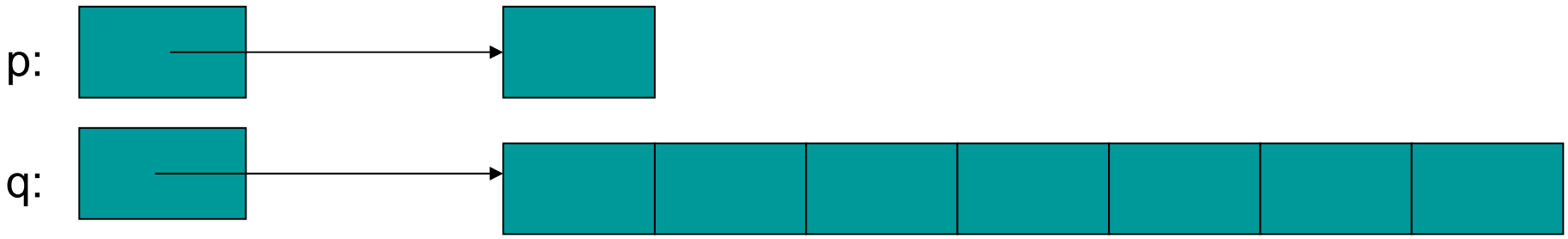


Pamięć wolna

- Do tworzenie obiektu (alokacja) w pamięci wolnej służy operator **new**
  - który zwraca wskaźnik do zaalokownego fragmentu pamięci np.

```
int* p = new int;              // alokuje pamięć na 1 liczbę całkowitą
                               // int* oznacza wskaźnik do int

int* q = new int[7];           // alokuje pamięć na 7 liczb całkowitych
                               // tj. tablicę siedmiu int-ów

double* pd = new double[n];    // alokuje pamięć na n liczb rzeczywist.
```

  - operator **new** nie inicjalizuje obiektów

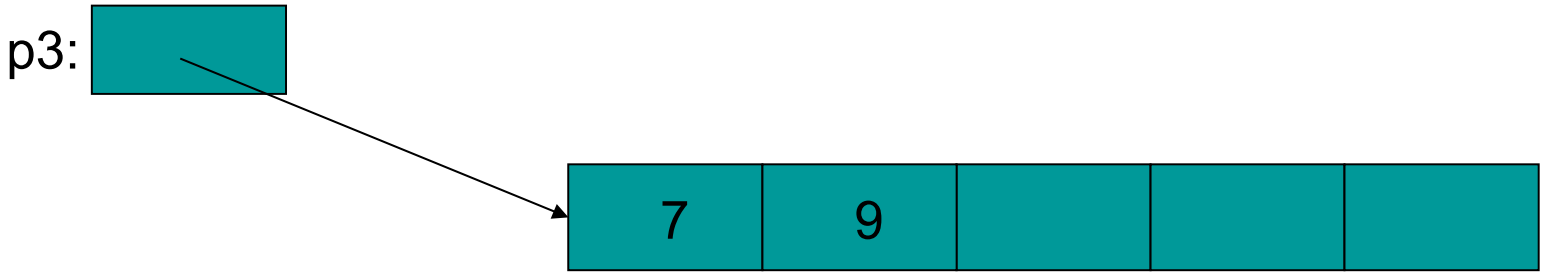


Dostęp do obiektów w pamięci wolnej

- Do odczytania wartośći obiektu w pamięci wolnej służy operator *, czyli operator dereferencji (wyłuskania), np.

```
int* p1 = new int;          // alokuje nowego niezainicjowanego int-a
int* p2 = new int(5);       // alokuje nowego int-a i inicjuje go wartością 5
int x = *p2;                // odczytuje wartość wskazywaną przez p2 (x wynosi 5)
int y = *p1;                // wynik niezdefiniowany, lepiej tego nie robić
```

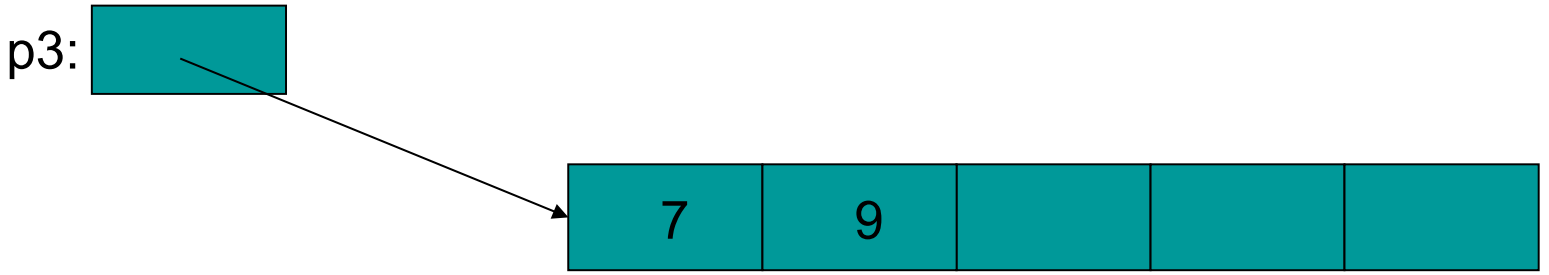
Dostęp do tablic w pamięci wolnej



- Do odczytania wartośći obiektu w pamięci wolnej służą operator dostępu [ ] oraz operator wyłuskania *, np.

```
int* p3 = new int[5];       // alokuje 5 niezainicjowanych int-ów

p3[0] = 7;          // zapis 1-go elementu p3
p3[1] = 9;

int x2 = p3[0];     // odczyt 1-go elementu p3
int x3 = p3[1];     // odczyt 2-go elementu p3
```

Dostęp do tablic w pamięci wolnej

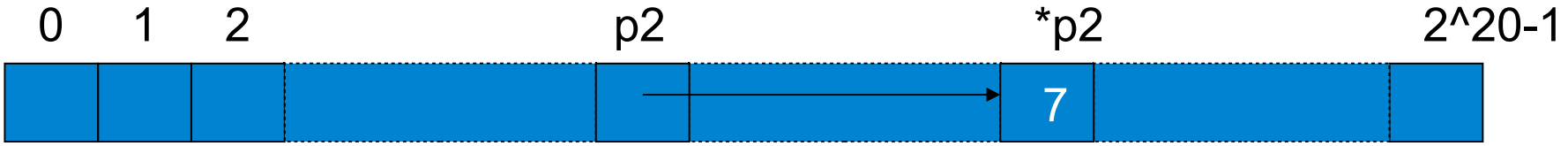


- Do odczytania wartośći obiektu w pamięci wolnej służą operator dostępu [ ] oraz operator wyłuskania *, np.

```
int* p3 = new int[5];        // alokuje 5 niezainicjowanych int-ów

*p3 = 7;        // zapis 1-go elementu p3
*(p3+1) = 9;

int x2 = *p3;        // odczyt 1-go elementu p3
int x3 = *(p3+1);// odczyt 2-go elementu p3
```

Wskaźniki

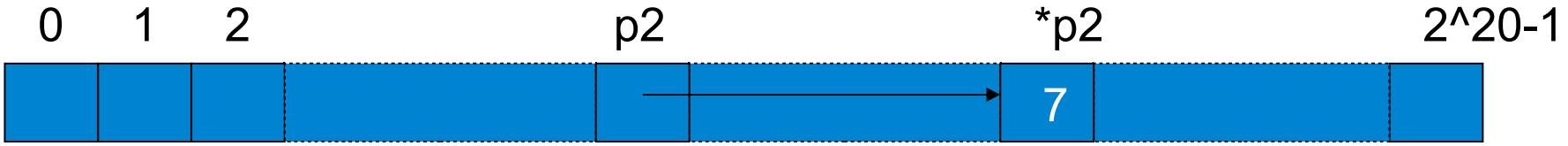
- Wskaźniki są adresami pamięci
  - można je traktować jak pewne wartości całkowite
  - pierwszy bajt pamięci ma adres 0, kolejny 1 itd.



- Wskaźnik wskazuje obiekt określonego typu (np. **int *p3**)
  - typ wskaźnika definiuje sposób korzystania z pamięci na którą wskazuje
  - typ wskaźnika pozwala określić adres kolejnego elementu w pamięci, np. ***(p3+1)** (==**p3[1]**), w zależności ile pamięci zajmuje element
- wskaźnik nie wie na ile elementów wskazuje!

Wskaźniki

- Wskaźniki są adresami pamięci
  - można je traktować jak pewne wartości całkowite
  - pierwszy bajt pamięci ma adres 0, kolejny 1 itd.



- Wartości wskaźników można wyświetlić (ale rzadko się to przydaje)

```
char c1 = 'c';
char* p1 = &c1;          // wskaźnik do obiektu na stosie
int* p2 = new int(7);    // wskaźnik do obiektu w pamięci wolnej

cout << "p1==" << p1 << " *p1==" << *p1 << "\n";  // p1==??? *p1=='c'
cout << "p2==" << p2 << " *p2==" << *p2 << "\n";  // p2==??? *p2=7
```

Wskaźnik a referencja

- czy znasz różnicę?

Dealokacja

- Obiekty w pamięci wolnej dealokuje operator **delete**
  - **delete** (dla pojedynczych obiektów) i **delete[ ]** (dla tablic) zwalnia pamięć zaalokowaną przez **new**

```
int* pi = new int;              // inicjacja domyślna (nieokreślona dla int)
char* pc = new char('a');       // inicjacja jawna
double* pd = new double[10];    // alokacja tablicy (niezainicjowanej )
delete pi;                      // dealokuje pojedyńczy obiekt
delete pc;                      // dealokuje pojedyńczy obiekt
delete[ ] pd;                   // dealokuje tablicę
```

- Dealokacja wskaźnika o adresie 0 (czyli *null*) niczego nie robi

```
char* p = 0;
delete p;                       // nieszkodliwe
```

Po co stosować pamięć wolną?

- Aby tworzyć obiekty o zmiennym rozmiarze (np. wektor, kolejka, drzewo...)

- Aby tworzyć obiekty, które będą żyły dłużej niż zakres w którym są tworzone, np.

```
double* make(int n)              // alokuje n int-ów
{
    return new double[n];
}
```

Problem wycieków pamięci

```
double* calc(int result_size, int max)
{
    double* p = new double[max];       // alokuj max obiektów double, tzn. zajmij
                                       // przestrzeń na max double-i w pamięci wolnej
                                       // oraz zapisz wskaźnik do pamięci w p
    double* result = new double[result_size];  // analogicznie dla result

    // ... użyj p do policzenia wyników, które zostaną umieszczone w pamięci
    // wskazywanej przez result ...

    return result;
}
double* r = calc(200,100);     // ups! Zapomnieliśmy oddać pamięć zaalokowaną
                               // dla p z powrotem na stertę
```

- Niby nic się nie stało, program zrobił swoje...
- A jednak brak dealokacji (czyli wyciek pamięci) może być powodem poważnych kłopotów w prawdziwym programie
  - szczególnie gdy program działa bardzo długo i wycieki się zbierają

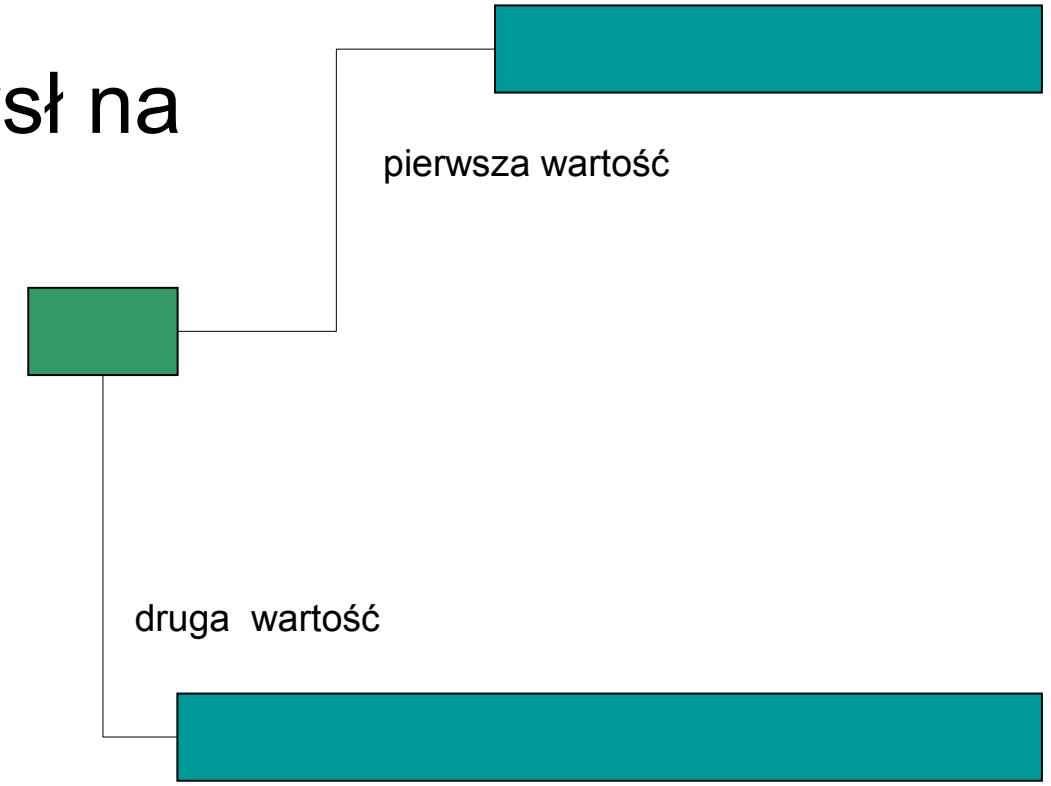
Problem wycieków pamięci (2)

```
double* calc(int result_size, int max)
{
    double* p = new double[max];      // alokuj max obiektów double, tzn. zajmij
                                      // przestrzeń na max double-i w pamięci wolnej
                                      // oraz zapisz wskaźnik do pamięci w p
    double* result = new double[result_size];  // analogicznie dla result

    // ... użyj p do policzenia wyników, które zostaną umieszczone w pamięci
    // wskazywanej przez result ...

    delete[ ] p;                      // dealokuje (zwalnia pamięć)
                                      // oddaje ją z powrotem na stertę
    return result;
}
double* r = calc(200,100);    // ups! Zapomnieliśmy oddać pamięć zaalokowaną
                              // dla p z powrotem na stertę
// korzystamy z r...
delete[ ] r;                  // łatwe do zapomnienia
```

Wycieki pamięci

- Jeszcze jeden pomysł na wyciek pamięci

```
void f()
{
    double* p = new double[27];
    // ...
    p = new double[42];
    // ...
    delete[] p;
}
```



*// pierwsza tablica (27 liczb **double**) wyciekła...*

O wyciekach pamięci

- Program, który chodzi w nieskończoność, nie może sobie pozwolić na żaden wyciek, np. system operacyjny
- Jeśli z funkcji wycieka 8 bajtów z każdym wywołaniem
  - po 130 000 wywołań stracimy 1 megabajt
- Cała pamięć jest zwracana systemowi operacyjnemu po zakończeniu programu (w Windowsie, Linuksie itp.)
- Dlatego często wycieki przechodzą niezauważone
  - ale w określonych okolicznościach mogą być bardzo groźne
- Gdy piszemy funkcję lub klasę nie wiemy, kto i w jaki sposób będzie z niej korzystał
  - dlatego musimy zawsze pilnować wycieków pamięci

Wskaźniki, tablice i **vector**

- Uwaga
  - Używając wskaźników i tablic dotykasz sprzętu przy minimalnym wsparciu języka
  - Stąd przy korzystaniu z nich łatwo o poważny i trudny do znalezienia błąd
  - Należy z nich korzystać tylko wtedy kiedy naprawdę są potrzebne
- **vector** jest jednym ze rozwiązań
  - by zachowując prawie całą elastyczność i wydajność tablic skorzystać z większego wsparcia języka (czytaj: mniej błędów, szybsze odrobaczanie)

Projektujemy **vector**

- **Założenia**
  - reprezentuje pojęcie tablicy jednowymiarowej
  - może przechowywać dowolną i zmienną liczbę elementów typu **double**
  - zapewnia inicjowanie elementów oraz dostęp do nich

```
// bardzo uproszczony vector liczb rzeczywistych typu double (por. vector<double>):
class vector {
    int sz;             // liczba elementów ("rozmiar", ang. size)
    double* elem;       // wskaźnik na pierwszy element
public:
    vector(int s);          // konstruktor: alokuj s elementów,
                            // niech elem wskazuje na nie
                            // zapisz s w sz
    int size() const { return sz; }     // zwróć bieżący rozmiar
};
```

Konstruktor **vector**-a

// bardzo uproszczony ***vector*** *liczb rzeczywistych typu* ***double*** *(por.* ***vector<double>****):*

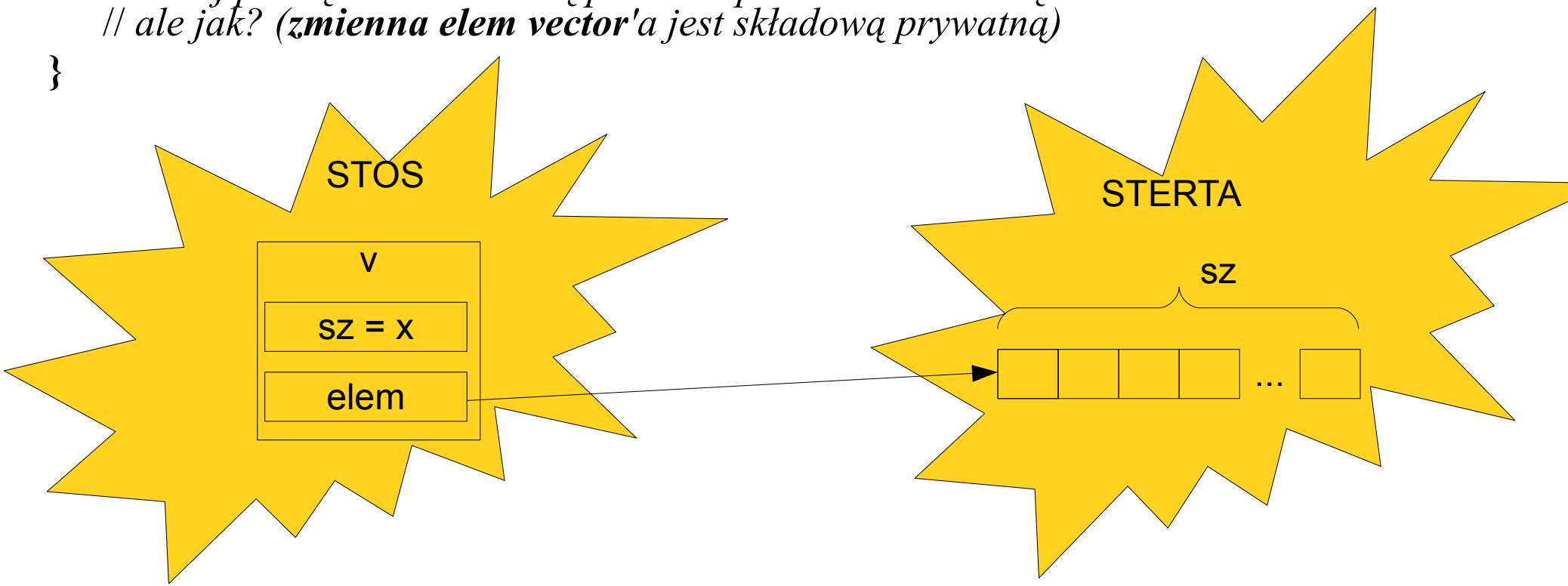
```
class vector {
    int sz;                 // liczba elementów (“rozmiar”, ang. size)
    double* elem;           // wskaźnik na pierwszy element
public:
    vector(int s);          // konstruktor: alokuj s elementów,
                            // niech elem wskazuje na nie
                            // zapisz s w sz
    int size() const { return sz; }     // zwróć bieżący rozmiar
};

vector::vector(int s)       // konstruktor wektora
    :sz(s),                 // zapisuje rozmiar s w sz
     elem(new double[s])    // alokuje s liczb typu double w wolnej pamięci
                            // zapisuje wskaźnik na te double w elem
{
}
```

// Operator ***new*** *alokuje pamięć wolną i zwraca wskaźnik do zaalokowanej pamięci.*
// Uwaga: ***new*** *nie inicjuje elementów (prawdziwy standardowy wektor inicjuje)*

Konstruktor **vector**-a

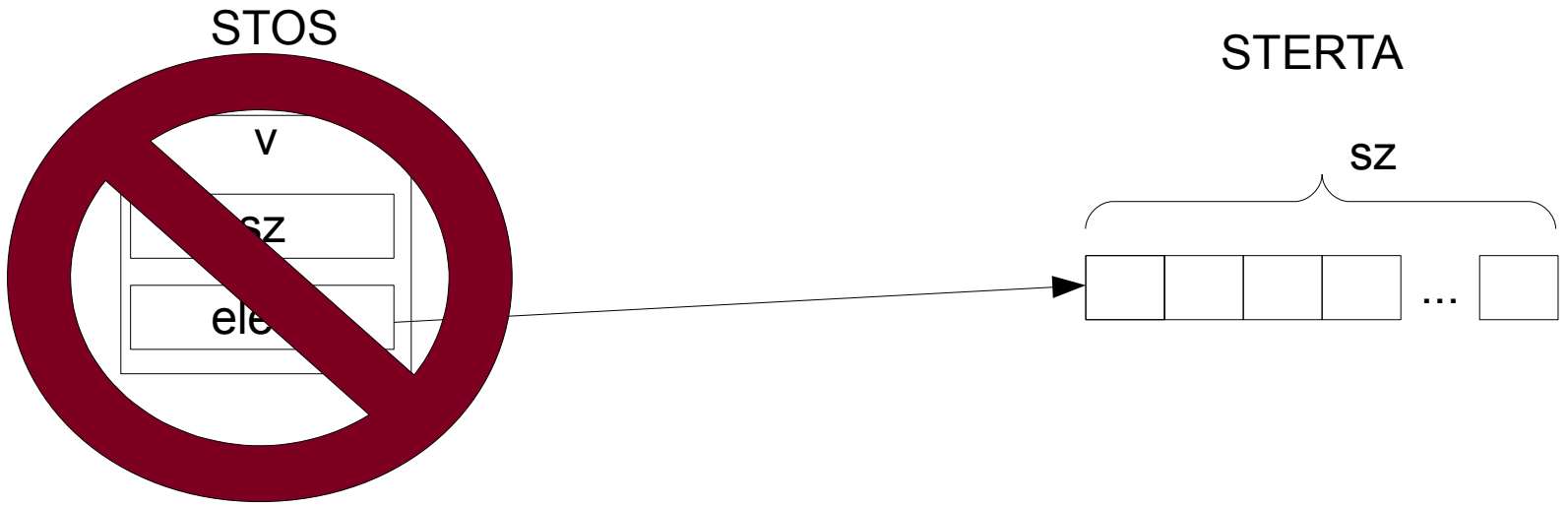
```
// bardzo uproszczony vector liczb rzeczywistych typu double (por. vector<double>):
class vector {
    int sz;                 // liczba elementów (“rozmiar”, ang. size)
    double* elem;           // wskaźnik na pierwszy element
public:
    vector(int s);          // konstruktor: alokuj s elementów,
                            // niech elem wskazuje na nie
                            // zapisz s w sz
    int size() const { return sz; }     // zwróć bieżący rozmiar
};

vector::vector(int s) {     // konstruktor wektora
    sz = s;                 // zapisuje rozmiar s w sz
    elem = new double[s];   // alokuje s liczb typu double w wolnej pamięci
                            // zapisuje wskaźnik na te double w elem

}
// Operator new alokuje pamięć wolną i zwraca wskaźnik do zaalokowanej pamięci.
// Uwaga: new nie inicjuje elementów (prawdziwy standardowy wektor inicjuje)
```

```
void f(int x)
{
    vector v(x);    // zdefiniuj vector v
                    // (który alokuje x double-i na stercie)
    // ... używaj v ...

    // oddaj pamięć zaalokowaną przez v z powrotem na stertę
    // ale jak? (zmienna elem vector'a jest składową prywatną)
}
```



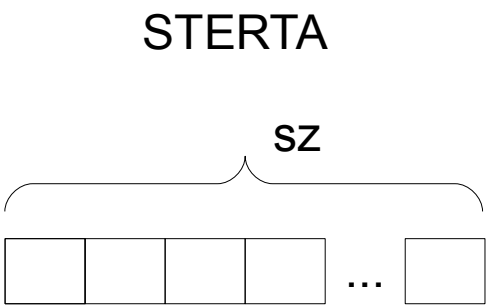
```
void f(int x)
{
    vector v(x);    // zdefiniuj vector v
                    // (który alokuje x double-i na stercie)
    // ... używaj v ...

    // oddaj pamięć zaalokowaną przez v z powrotem na stertę
    // ale jak? (zmienna elem vector'a jest składową prywatną)
}
```

```
void f(int x)
{
    vector v(x);    // zdefiniuj vector v
                    // (który alokuje x double-i na stercie)
    // ... używaj v ...

    // oddaj pamięć zaalokowaną przez v z powrotem na stertę
    // ale jak? (zmienna elem vector'a jest składową prywatną)
}
```



Problem: wyciek pamięci

```
void f(int x)
{
    vector v(x);    // zdefiniuj vector v
                    // (który alokuje x double-i na stercie)
    // ... używaj v ...

    // oddaj pamięć zaalokowaną przez v z powrotem na stertę
    // ale jak? (zmienna elem vector'a jest składową prywatną)
}
```



!

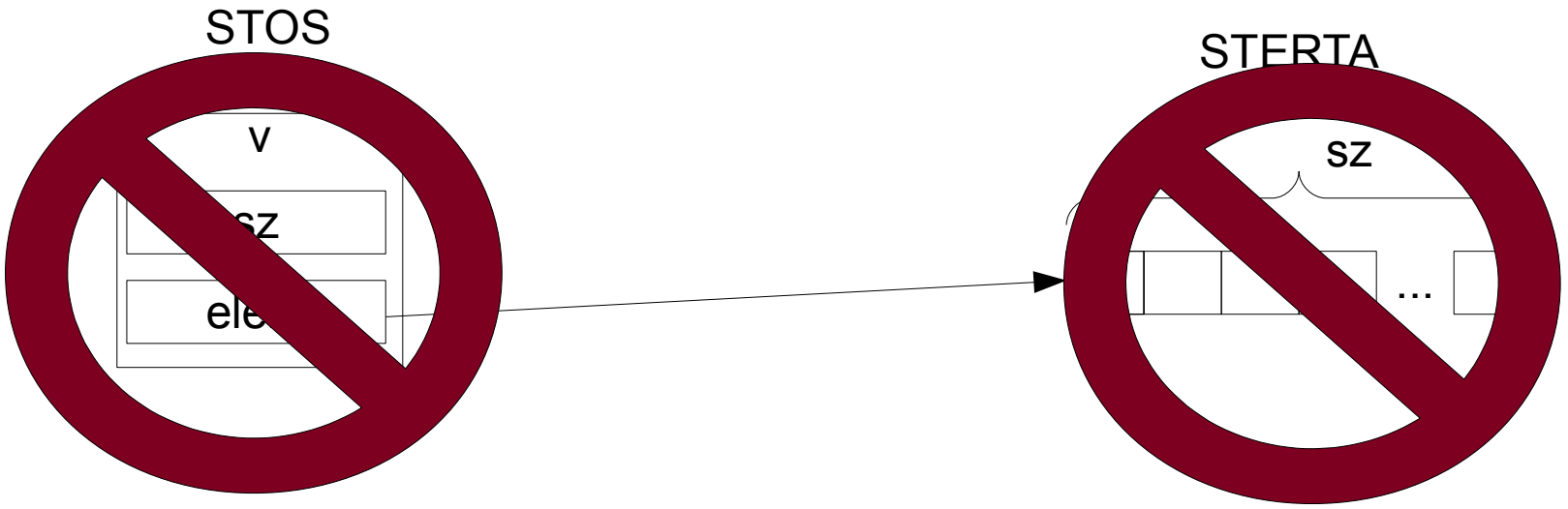
Destruktor **vector**-a

```
// bardzo uproszczony vector liczb rzeczywistych typu double
class vector {
    int sz;             // rozmiar
    double* elem;       // wskaźnik na tablicę elementów
public:
    vector(int s)       // konstruktor: alokuj/zajmuje pamięć
    :sz(s), elem(new double[s]) {}
    ~vector()           // destruktor: dealokuje/zwalnia pamięć
    { delete[ ] elem; }
    //...
};
```

- To jest przykład ogólnej i ważnej techniki
  - konstruktor zajmuje zasoby
  - destruktor zwalnia je
- Przykłady zasobów:
  - pamięć, pliki, blokady, wątki, gniazda

Unikanie wycieków pamięci przy użyciu **vectora**

```
void f(int x)
{
    vector v(x);                    // zdefiniuj vector v
                                    // (który alokuje x double-ów na stercie)
    // używaj v ...
}   // pamięć alokowana przez v jest niejawnie czyszczona przez destruktor vectora
```



Unikanie wycieków pamięci przy użyciu **vectora**

```
void f(int x)
{
    vector v(x);                        // zdefiniuj vector v
                                        // (który alokuje x double-ów na stercie)
    // używaj v ...
}   // pamięć alokowana przez v jest niejawnie czyszczona przez destruktor vectora

void f(int x)
{
    double* p = new double[x];      // alokuj x double-ów
    // używaj p ...
    delete[ ] p;     // dealokacja tablicy wskazywanej przez p
}
```

- Podejście z **delete** wygląda brzydko i rozwlekle
  - jak uniknąć zapomnienia **delete[ ] p**?
  - doświadczenie uczy, że zapomina się często
- Lepiej, gdy **delete** jest w destruktorze

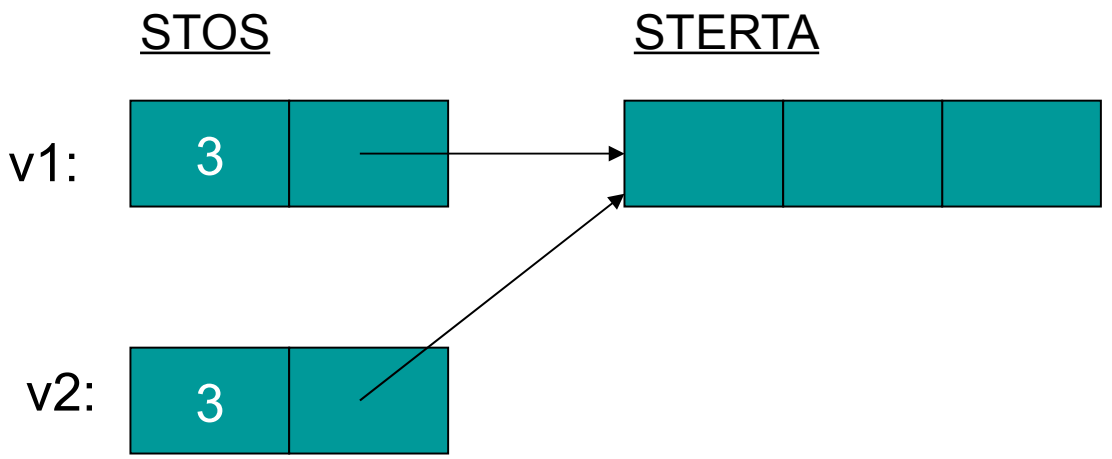
Kopiowanie

```
void f(int n)
{
    vector v(n);        // definiuje vector
    vector v2 = v;      // co dzieje się tutaj?
                        // a co byśmy chcieli żeby się działo?
    vector v3;
    v3 = v;             // co dzieje się tutaj?
                        // a co byśmy chcieli żeby się działo?
    // ...
}
```

- Idealnie: **v2** i **v3** powinny stać się kopiami **v** (czyli chcemy aby operator = wykonywał kopię)
  - A cała pamięć powinna zostać zwrócona na stertę w chwili opuszczenia **f()**
- Tak działa **vector** w bibliotece standardowej
  - a jak działa nasz wciąż zbyt uproszczony wektor?

Domyślna inicjacja kopii jest naiwna

```
void f(int n)
{
    vector v1(n);
    vector v2 = v1;          // inicjacja:
                             // domyślnie kopiowane są składowe klasy
                             // czyli kopiowane są sz i elem
}
```



Kiedy wyjdziemy z funkcji **f()** czeka nas katastrofa!

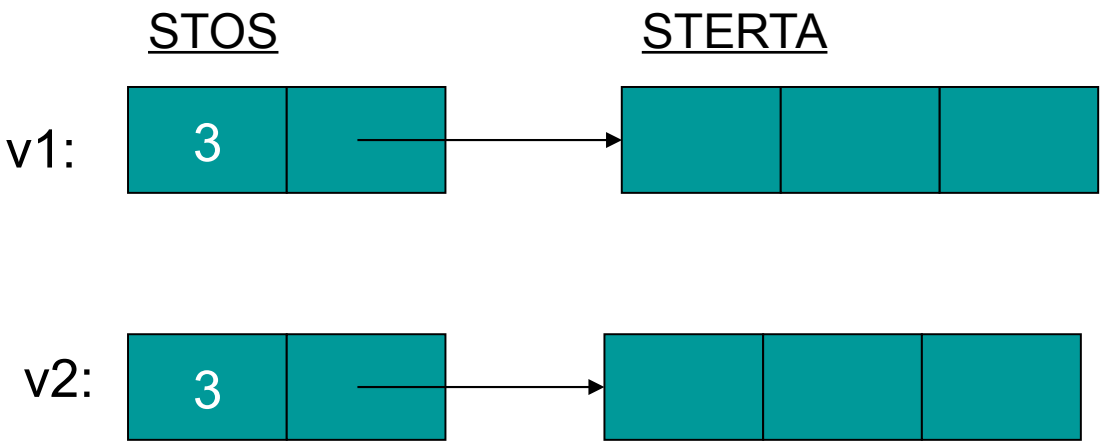
- elementy wektora **v1** są dwukrotnie usuwane przez destruktor

Musimy wykonać kopię elementów: tworzymy **konstruktor kopiujący**

```
class vector {
    int sz;
    double* elem;
public:
    vector(const vector&) ;          // konstruktor kopiujący klasy vector
    // …
};

vector::vector(const vector& a)
    :sz(a.sz),  elem(new double[a.sz])
    // alokuje przestrzeń na elementy wektora, potem inicjuje je przez kopiowanie
{
    for (int i = 0; i<sz; ++i) elem[i] = a.elem[i];
}
```

Kopiowanie z użyciem konstruktora kopiującego

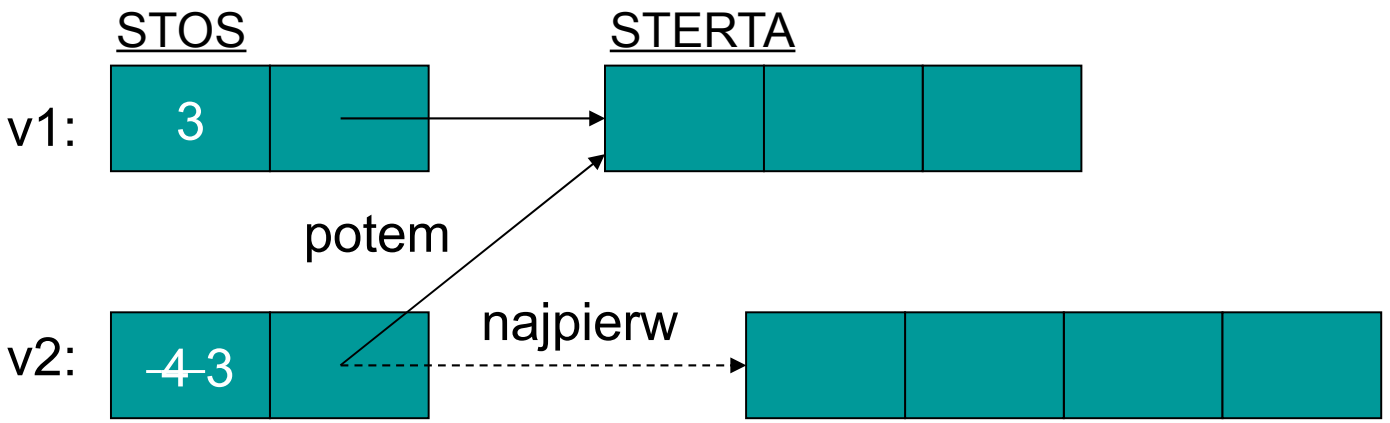
```
void f(int n)
{
    vector v1(n);
    vector v2 = v1;          // kopiowanie z użyciem konstruktora kopiującego
                             // pętla for kopiuje każdą wartość v1 do v2
}
```



Destruktor poprawnie usuwa wszystkie elementy

Domyślny operator przypisania jest naiwny

```
void f(int n)
{
    vector v1(n);
    vector v2(4);
    v2 = v1;        // przypisanie :
                    // domyślnie podmieniane są składowe klasy
                    // czyli podmieniane są sz i elem
}
```

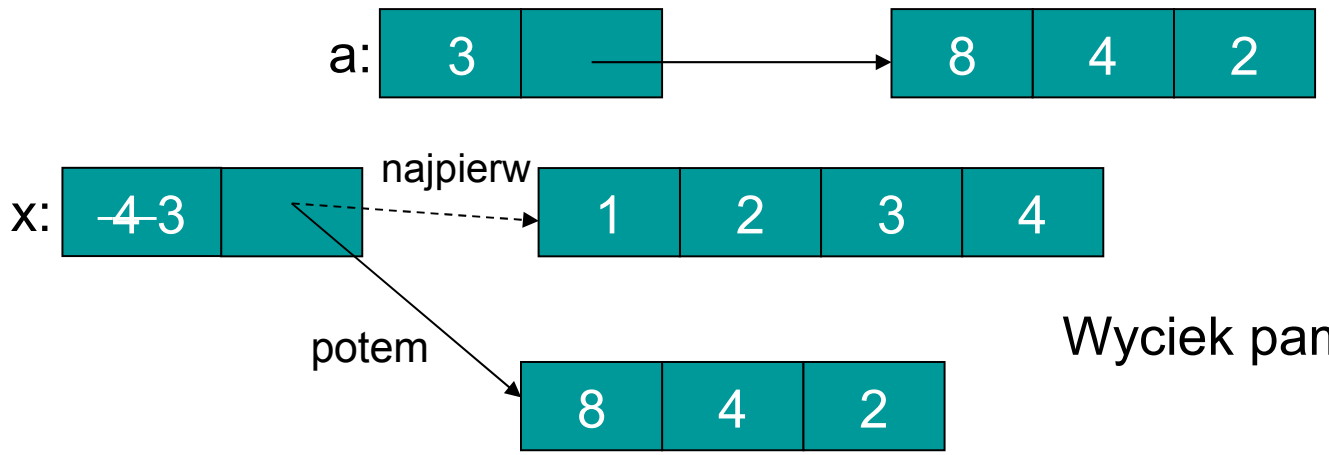


Kiedy wyjdziemy z funkcji **f()** czeka nas katastrofa!

- elementy wektora **v1** są dwukrotnie usuwane przez destruktor
- oryginalne elementy wektora **v2** nie są w ogóle usuwane

Operator przypisania – koncepcja

```
class vector {
    int sz;
    double* elem;
public:
    vector& operator=(const vector& a);    // operator przypisania
    // ...
};

x=a;
```

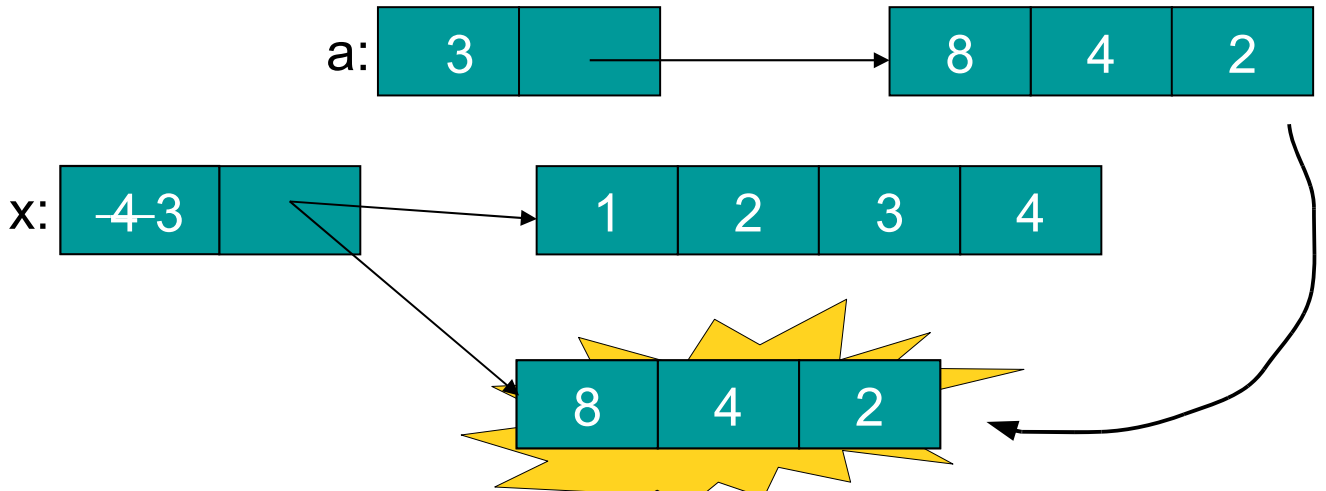


Wyciek pamięci? (nie)

Operator = musi kopiować elementy **a** do **x**

Operator przypisania – implementacja

```
vector& vector::operator=(const vector& a)
    // podobny do konstruktora kopiującego, ale musi coś zrobić ze starymi elementami
    // wykonuje kopię a, potem zamienia bieżący sz i elem z tymi z a
{
    double* p = new double[a.sz];          // alokuje nową przestrzeń
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];   // kopiuje elementy
    delete[ ] elem;                        // dealokuje starą przestrzeń
    sz = a.sz;                             // ustawia nowy rozmiar
    elem = p;                              // ustawia nowe elementy
    return *this;
}
```



Operator przypisania – implementacja

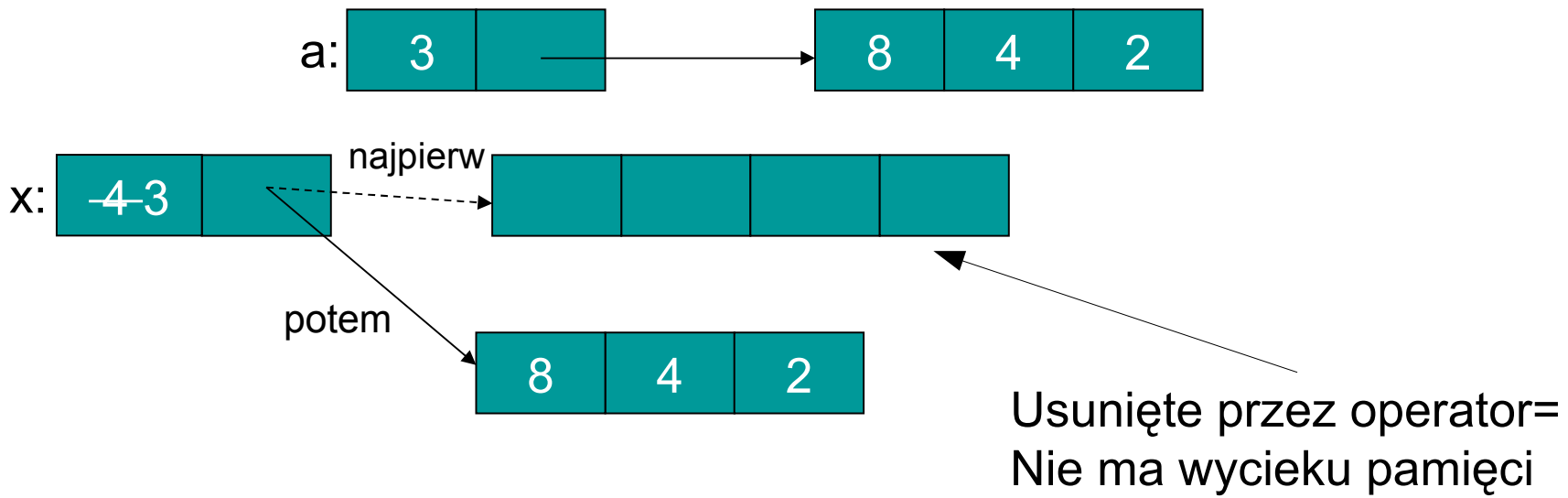
```
vector& vector::operator=(const vector& a)
    // podobny do konstruktora kopiującego, ale musi coś zrobić ze starymi elementami
    // wykonuje kopię a, potem zamienia bieżący sz i elem z tymi z a
{
    double* p = new double[a.sz];          // alokuje nową przestrzeń
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];   // kopiuje elementy
    delete[ ] elem;                        // dealokuje starą przestrzeń
    sz = a.sz;                             // ustawia nowy rozmiar
    elem = p;                              // ustawia nowe elementy
    return *this;
}
```

- Na marginesie
  zwracamy referencję do lewej strony przypisania – umożliwia sekwencję przypisań, jak dla typów wbudowanych, np.

```
int a, b, c, d, e;
a = b = c = d = e = 153;        // Interpretowane jako: a = (b = (c = (d = (e = 153))));
```

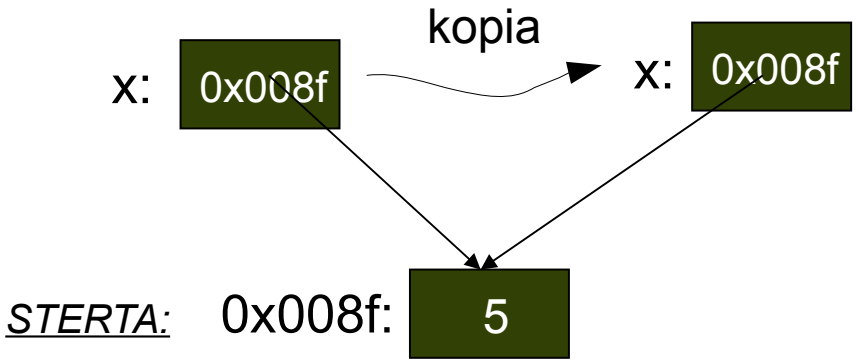
Kopiowanie przez przypisanie

```
void f(int n)
{
    vector v1(n);
    vector v2(4);
    v2 = v1;                    // przypisanie
}
```

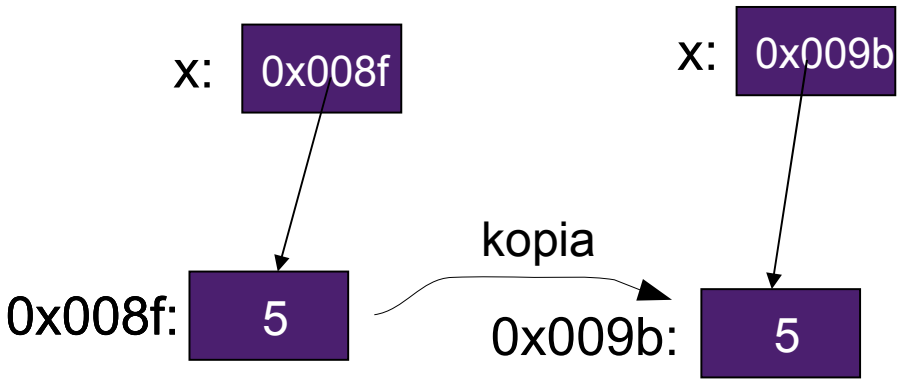


Terminologia kopiowania

- Płytka kopia (ang. *shallow copy*):
  - **kopiuje tylko wskaźnik**, w efekcie dwa wskaźniki odnoszą się do tego samego obiektu
  - tak działa kopiowanie wskaźników i referencji
- Głęboka kopia (ang. *deep copy*)
  - **kopiuje to na co wskazuje wskaźnik i przestawia wskaźnik**, w efekcie dwa wskaźniki odnoszą się do dwóch osobnych obiektów
  - tak działa kopiowanie **vector**-ów, **string**-ów itp.
  - Wymaga konstruktora kopiującego i operatora przypisania



Płytka kopia



Głęboka kopia

Sprawdzanie zakresu

// prawie prawdziwy fragment implementacji ***vector****a:*

```
struct out_of_range { /* ... */ };
class vector {
    // ...
    double& operator[ ](int n);     // operator dostępu
    double& at(int n);              // funkcja dostępu ze sprawdzaniem zakresu
    // ...
};

double& vector::operator[ ](int n) {      // nie sprawdza zakresu, więc trochę szybsze
    return elem[n];                       // tam gdzie liczy się optymalna wydajność
                                          // np. urządzenia czasu rzeczywistego
}                                         // nie stosujemy obsługi wyjątków

double& vector::at(int n) {   // średnio 3% gorsza wydajność
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}
```

Obsługa wyjątków (prymitywna)

```
// czasami zapewnienie porządku po błędzie wymaga trochę pracy

void fill_vec(vector& v, int n);

vector* some_function()  // tworzy wypełniony vector
{
    vector* p = new vector;          // alokuje na stercie
                                     // ktoś musi dealokować
    try {
        fill_vec(*p,10);
        // ...
        return p;          // wypełnianie poszło dobrze, zwracamy pełny wektor
    }
    catch (…) {            // coś poszło źle:
        delete p;          // robimy lokalny porządek
        throw;             // ponownie zgłaszamy wyjątek, aby funkcja wywołująca
                           // mogła dokończyć rozwiązywanie problemu
    }
}
```

Obsługa wyjątków
(prostsza i bardziej czytelna)

```
// Kiedy używamy zmiennych w zakresie, porządek jest robiony automatycznie

void fill_vec(vector& v, int n);

vector some_other_function()      // tworzy wypełniony vector
{
    vector v;        // uwaga: vector obsługuje dealokację elementów

    fill_vec(v,10);
    // użycie v
    return v;
}
```

- Wniosek: jeśli czujesz, że potrzebujesz bloku try-block: pomyśl
  - być może możesz sobie poradzić (lepiej) bez tego

Zarządzanie zasobami przez zakres
czyli zajęcie zasobów jest inicjacją

- **vector**
  - zajmuje pamięć na elementy w swoim konstruktorze
  - zarządza nią (zmiana rozmiaru, kontrola dostępu itp.)
  - zwalnia pamięć w destruktorze
- To przypadek szczególny ogólnej strategii zarządzania pamięcią zwanej **RAII,** czyli ***Resource Acquisition Is Initialization***
  - zwanej także zarządzaniem zasobami przez zakres
  - używaj je zawsze, gdy to możliwe
  - prostsze i oszczędniejsze niż cokolwiek innego
  - świetnie współpracuje z obsługą błędów przez wyjątki

Sprytny wskaźnik **unique_ptr** (C++11)

```
// Kiedy używamy zmiennych w zakresie, porządek jest robiony automatycznie

#include <memory>
void fill_vec(vector& v, int n) ;
vector* another_other_function()      // tworzy wypełniony vector
{
    std::unique_ptr<vector> p(new vector);
                                      // umieszcza wektor na stercie
                                      // tworzy do niego sprytny wskaźnik
    fill_vec(*p,10);
    // użycie p
    return p.release();               // zwraca zwykły wskaźnik do wektora
                                      // jednak jeśli operator new, funkcja fill_vec
                                      // lub cokolwiek innego zgłosi wyjątek,
                                      // to sprytny wskaźnik zostanie usunięty z pamięci
                                      // razem z wektorem
}
```

- Sprytny wskaźnik to klasa symulująca zwykły wskaźnik
  - m.in. poprzez przeładowanie operatorów -> oraz *
  - plus dodatkowa funkcjonalność, np. odśmiecanie albo kontrola zakresów
  - możesz pisać swoje własne sprytne wskaźniki

Dalej oglądamy **vector**

- Zasadniczy problem
  - chcemy by wektor zmieniał rozmiar, gdy zmieniamy liczbę elementów
  - ale w pamięci komputera wszystko musi mieć określony rozmiar
  - jak stworzyć iluzję, że rozmiar się zmienia?
- Mając

  vector v(n); // *v.size()==n*

  możemy zmienić jego rozmiar na trzy sposoby:

  v.resize(10); // ***v** ma teraz 10 elementów*

  v.push_back(7); // *dodaje element o wartości 7 na koniec **v***
  // ***v.sz** zwiększa się o 1*

  v = v2; // ***v** jest teraz kopią **v2***
  // ***v.size()** jest teraz równe **v2.size()***

Reprezentacja wektora

- Obserwacja:
  - jeśli zmieniasz rozmiar (**resize()** lub **push_back()**) raz, zapewne zrobisz to powtórnie
  - przygotujmy się na to rezerwując trochę wolnej przestrzeni na przyszłość

```
class vector {
    int sz;
    double* elem;
    int space;      // liczba elementów plus "wolna przestrzeń"
                    // (czyli liczba komórek na nowe elementy)
public:
    // …
};
```

Zacznijmy od rezerwacji pamięci

- Zauważ:
  - rezerwacja nie zmienia rozmiaru wektora ani wartości jego elementów
  - definicja funkcji **reserve()** gwarantuje, że w razie wystąpienia wyjątku (operator **new**) stan obiektu pozostanie niezmieniony i nie wycieknie żadna pamięć. Jest to tzw. silna gwarancja

```
void vector::reserve(int newalloc)
// tworzy wektor który ma przestrzeń na newalloc elementów
{
    if (newalloc<=space) return;              // nigdy nie zmniejsza alokacji
    double* p = new double[newalloc];         // alokuje nową przestrzeń
    for (int i=0; i<sz; ++i) p[i]=elem[i];    // kopiuje elementy
    delete[ ] elem;                           // de-alokuje starą przestrzeń
    elem = p;
    space = newalloc;
}
```

Teraz zmiana rozmiaru jest prosta

- **reserve()** zajmuje się alokacją przestrzeni
- **resize()** zajmuje się wartościami elementów
- Znów otrzymujemy silną gwarancję bezpieczeństwa wyjątków

```
void vector::resize(int newsize)
// tworzy wektor, który ma newsize elementów
// inicjuje każdy nowy element domyślną wartością 0.0
{
    reserve(newsize);                          // upewnia się, że mamy dość przestrzeni
    for(int i = sz; i<newsize; ++i) elem[i] = 0.0;        // inicjuje nowe elementy
    sz = newsize;
}
```

Dodawanie na koniec też jest proste

- **reserve()** zajmuje się alokacją przestrzeni
- **push_back()** tylko dodaj jedną wartość
- Znów otrzymujemy silną gwarancję bezpieczeństwa wyjątków

```
void vector::push_back(double d)
// zwiększa rozmiar wektora o 1
// inicjuje nowy element wartością d
{
    if (sz==0)                  // wektor pusty: zaalokuj trochę przestrzeni
            reserve(8);
    else if (sz==space)         // brakuje przestrzeni: zaalokuj trochę więcej
            reserve(2*space);
    elem[sz] = d;               // dodaj d na koniec
    ++sz;                       // i zwiększ rozmiar
}
```

resize() and push_back()

```
class vector {                  // an almost real vector of doubles
    int sz;                     // the size
    double* elem;               // a pointer to the elements
    int space;                  // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }                  // default constructor
    explicit vector(int s) :sz(s), elem(new double[s]) , space(s) { }    // constructor
    vector(const vector&);                                   // copy constructor
    vector& operator=(const vector&);                        // copy assignment
    ~vector() { delete[ ] elem; }                            // destructor

    double& operator[ ](int n) { return elem[n]; }           // access: return reference
    int size() const { return sz; }                          // current size

    void resize(int newsize);                                // grow (or shrink)
    void push_back(double d);                                // add element

    void reserve(int newalloc);                              // get more space
    int capacity() const { return space; }                   // current available space
};
```

Operator przypisania

- Stosujemy ogólną technikę “kopiuj i zamień”
- Mamy silną gwarancję bezpieczeństwa wyjątków

```
vector& vector::operator=(const vector& a)
    // podobny do konstruktora kopiującego, ale musi coś zrobić ze starymi elementami
    // wykonuje kopię a, potem zamienia bieżący sz i elem z tymi z a
{
    double* p = new double[a.sz];          // alokuje nową przestrzeń
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];   // kopiuje elementy
    delete[ ] elem;                        // dealokuje starą przestrzeń
    sz = a.sz;                             // ustawia nowy rozmiar
    space = a.sz;                          // ustawia nową pojemność
    elem = p;                              // ustawia nowe elementy
    return *this;
}
```

Operator przypisania – zoptymalizowany

- Jeśli dość miejsca na przypisywany wektor – nie trzeba alokować nowej pamięci
- Tracimy jednak silną gwarancję bezpieczeństwa wyjątków
  - wyjątek w trakcie kopiowania spowoduje, że stan obiektu będzie pośredni między początkowym a docelowym
  - ale będzie poprawny, nie nastąpi też wyciek pamięci (tzw. podstawowa gwarancja)

```
vector& vector::operator=(const vector& a) {
    if (this==&a) return *this;          // czy nie ma autoprzypisania
    if (a.sz<=space) {                   // dość miejsca, nie trzeba alokować
        for (int i = 0; i<a.sz; ++i) elem[i] = a.elem[i];   // kopiuje elementy
        space += sz-a.sz;                                   // zwiększa licznik wolnej przestrzeni
        sz = a.sz;                                          // ustawia nowy rozmiar
        return *this;
    }
    double* p = new double[a.sz];                           // alokuje nową przestrzeń
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];          // kopiuje elementy
    delete[ ] elem;                                         // dealokuje starą przestrzeń
    sz = a.sz;                                              // ustawia nowy rozmiar
    space = a.sz;                                           //          nową pojemność
    elem = p;                                               //          nowe elementy
    return *this;
}
```

Dziś najważniejsze było...

- Unikamy bezpośredniej pracy z pamięcią wolną
- Kopia płytka i głęboka

  a obiekty z dynamicznie alokowaną pamięcią

- Technika zarządzania zasobami przez zakres (RAII)
- Gwarancje bezpieczeństwa wyjątków
  - więcej: http://www2.research.att.com/~bs/3rd_safe.pdf lub: http://www2.research.att.com/~bs/except.pdf

Dla zaawansowanych

Gdzie żyją obiekty


```
vector glob(10);                    // vector globalny – "żyje" zawsze

vector* some_fct(int n)
{
    vector v(n);                    // vector lokalny – "żyje" do końca zakresu
    vector* p = new vector(n);      // vector na stercie – "żyje" aż nie usunięty przez delete
    // …
    return p;
}

void f()
{
    vector* pp = some_fct(17);
    // …
    delete pp;      // dealokacja vectora w wolnej pamięci, zaalokowanego przez some_fct()
}
```

- łatwo zapomnieć usunąć obiekt zaalokowany w pamięci wolnej
  - unikaj **new**/**delete** jeśli możesz

A za 2 tygodnie...

- Dziedziczenie

Listy inicjacyjne C++11

Dla ciekawych

```
vector<double> v = { 1, 2, 3.456, 99.99 };

template<class E> class vector {
public:
        vector (std::initializer_list<E> s) // initializer-list constructor
        {
                reserve(s.size());        // get the right amount of space
                uninitialized_copy(s.begin(), s.end(), elem);
                                          // initialize elements (in elem[0:s.size()))
                sz = s.size();       // set vector size
        }

        // ... as before …
};

vector<double> v1(7);       // ok: v1 has 7 elements
v1 = 9;                     // error: no conversion from int to vector
vector<double> v2 = 9;      // error: no conversion from int to vector

vector<double> v1{7};             // ok: v1 has 1 element (with its value 7)
v1 = {9};                         // ok v1 now has 1 element (with its value 9)
vector<double> v2 = {9};          // ok: v2 has 1 element (with its value 9)


int x0 {7.3};       // error: narrowing
int x1 = {7.3};     // error: narrowing; the = is optional

double d = 7;
int x2{d};                                // error: narrowing (double to int)
char x3{7};                               // ok: even though 7 is an int, this is not narrowing
vector<int> vi = { 1, 2.3, 4, 5.6 };      // error: double to int narrowing
```

http://www2.research.att.com/~bs/C++0xFAQ.html#init-list