Algorytmy i struktury danych

Wstęp, wskaźniki

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Zasady

- E-nauczanie: Algorytmy i Struktury Danych (zaoczne)
- hasło ...
- wykład (50%):
  - egzamin pisemny
  - min. 50% do zaliczenia wykładu
- projekt (50%):
  - maks. 105 pkt. (programy do napisania + odpowiedź z kodu)
  - min. 50pkt. (=50%) do zaliczenia projektu
  - oddawanie zadań na zajęciach projektowych
  - kilka mniejszych zadań po 5 pkt. (oceniane binarnie), maks. 25 pkt.
  - kilka większych zadań po 10-25 pkt.
- min. 50% z całości

Projekt

- http://stos.eti.pg.gda.pl
- proszę założyć konta:
  - login: *numer-indeksu* np. 123456
  - hasło: dowolne, co najmniej 6 znaków
  - komentarz: imię, nazwisko
- konta wymagają mojej akceptacji (której konto nie uzyska jeżeli nie spełnia ww. warunków)
- przed rozpoczęciem projektu proszę utrwalić sobie informacje na temat podstaw programowania (zmienne, pętle, instrukcje warunkowe itp.), wczytywania danych z klawiatury i wypisywania na ekran

Projekt

- oddanie zadania projektowego wymaga zgłoszenia go w systemie w terminie oraz odpowiedzi z kodu na zajęciach; w przypadku awarii kod należy przesłać mailem lub przynieść na zajęcia; awarie nie wpływają na zmianę terminu
- korzystanie z STLa jest zabronione; dotyczy to wszystkich klas i funkcji z przestrzeni nazw std (w tym std::string), poza obiektami cin, cout i cerr; wykorzystanie zabronionych klas/funkcji powoduje brak punktów za zadanie,
- wymagana jest całkowicie samodzielna praca

Projekt

- rozwiązanie zadania to nie tylko napisanie kodu źródłowego, ale także wymyślenie rozwiązania i samodzielne rozwiązanie problemów, które wystąpią w trakcie implementacji, niesamodzielna praca (wliczając wszelkie "kolega pomagał" i korepetycje) skutkuje niezaliczeniem projektu i brakiem ewentualnego częściowego zaliczenia wykładu;
- wynik procentowy prezentowany przez system nie jest wyznacznikiem oceny a jedynie przedstawia, jaki procent testów w systemie zostało zaliczonych; na liczbę zdobytych punktów mają wpływ: odpowiedź z kodu, sposób rozwiązania (wykorzystany algorytm i struktury danych) oraz zgodność z zasadami,
- brak zwalniania pamięci w zadaniu powoduje utratę 15% maksymalnej liczby punktów za zadanie,
- dodawanie na koniec listy jednokierunkowej w czasie O(n) powoduje utratę 15% maksymalnej liczby punktów za zadanie,

Projekt

- zadania mogą narzucać dodatkowe wymagania (przedstawione w treści zadania) które mogą, ale nie muszą być automatycznie weryfikowane; niespełnienie takich wymagań skutkuje brakiem punktów za zadanie lub pojedynczy test; przykładowe ograniczenia to: limit czasu na wykonanie jednego testu (automatycznie sprawdzany), limit wykorzystanej pamięci (automatycznie sprawdzany), zastosowanie konkretnego algorytmu (nie jest automatycznie sprawdzane), zakaz stosowania wskazanych funkcji (nie jest automatycznie sprawdzane),
- liczba zgłoszeń rozwiązania nie ma wpływu na ocenę,
- oceniane jest jedynie ostatnie zgłoszenie,

Literatura

- T.Cormen i in. "Wprowadzenie do algorytmów"
- L.Banachowski i in. "Algorytmy i struktury danych"
- N.Wirth "Algorytmy + Struktury danych = programy"
- L.Banachowski i in. "Analiza algorytmów i struktur danych"
- M.Sysło i in. "Algorytmy optymalizacji dyskretnej"
- K. Goczyła "Struktury danych"

Formatowanie

- zwykły tekst
- `fragment kodu programu`
- `wynik działania fragmentu kodu`
- 1: pseudokod programu
  2: **while** nie koniec **do**
  3: zrób coś
  4: zrób coś jeszcze
  5: nic nie rób
  6: **end while**
  7: **return** “zrobione”

Wejście/wyjście — przypomnienie

- wczytywanie danych z klawiatury:
  - `cin >> zmienna`
  - `scanf("%X", &zmienna);` gdzie X zależy od typu zmiennej, np.:

```
int i; scanf("%d", &i);
double d; scanf("%lf", &d);
char c; scanf("%c", &c);
char txt[128]; scanf("%s", &txt);
```

- obie metody pomijają wszystkie początkowe białe znaki (spacje, nowe linie, tabulacje) a następnie wczytują to, co im odpowiada; kończą w momencie wystąpienia niepasującego znaku
- UWAGA: zasada ta nie dotyczy wczytywania pojedynczych znaków przy pomocy scanf: `scanf("%c", &znak);` przeczyta pojedynczą spację

Wejście/wyjście — przypomnienie

- czytając liczbę scanf/cin i wpisując “12” odczytamy liczbę 12
- czytając liczbę scanf/cin i wpisując “␣␣␣12␣␣␣” odczytamy liczbę 12
- czytając liczbę scanf/cin i wpisując “␣␣12XYZ” odczytamy liczbę 12
- czytając liczbę scanf/cin i wpisując “␣␣XYZ12” nie odczytamy żadnej liczby

Wejście/wyjście — przypomnienie

- scanf zwraca liczbę zmiennych, które udało mu się wypełnić
- w przypadku końca pliku (Ctrl+Z / Ctrl+D przy wpisywaniu z klawiatury) scanf zwraca -1
- wczytywanie dopóki są wprowadzane jakieś liczby:
  `while(scanf("%d", &liczba) > 0) ...`
- strumień cin rzutowany na wartość logiczną określa, czy ostatnia operacja powiodła się (true) czy nie (false)
- wczytywanie dopóki są wprowadzane jakieś liczby:
  `while(cin >> liczba) ...`

Wejście/wyjście — przypomnienie

- wypisywanie danych na ekran:
  - `cout << zmienna;`
  - `printf("%X", zmienna);` gdzie X to zależy od typu zmiennej, np.:

```
int i; ...  printf("%d", i);
double d; ...  printf("%lf", d);
char c; ...  printf("%c", c);
char txt[128]; ...  printf("%s", txt);
```

- UWAGA: scanf — ze znakiem &, printf — bez znaku & (wyjaśni się to przy wskaźnikach)

Wejście/wyjście — przypomnienie

- `cout << 12 << 34;` wypisze:

```
1234
```

- `printf("%d%d", 12, 34);` wypisze:

```
1234
```

- `cout << 12; cout << 34;` wypisze:

```
1234
```

- `printf("%d", 12); printf("%d", 34);`
  wypisze:

```
1234
```

Wejście/wyjście — przypomnienie

- `cout << 12 << ' ' << 34;` wypisze:

```
12 34
```

- `printf("%d %d", 12, 34);` wypisze:

```
12 34
```

- `cout << 12; cout << ' ' << 34;` wypisze:

```
12 34
```

- `printf("%d", 12); printf(" %d", 34);` wypisze:

```
12 34
```

- `cout << "\n";` to niezupełnie to samo co `cout << endl;`

Struktury (jeszcze nie danych)

- struktura to definiowany przez programistę nowy, złożony typ danych
- po deklaracji nowego typu możemy używać go na takich samych zasadach jak typów wbudowanych (int, double, char itp.)
- złożony — struktura jest zbiorem nazwanych pól, gdzie każde pole jest zmienną pewnego typu

Struktury

- deklaracja struktury ma postać:

```
struct Nazwa {
      Typ1 pole1;
      Typ2 pole2, pole3;
      ...
      TypN poleN;
      };
```

Struktury

- np. struktura reprezentująca datę może mieć postać:

```
struct Data {
      int rok;
      int miesiac;
      int dzien;
      };
```

Struktury

- zanim użyjemy struktury, musimy ją zdefiniować (poprzednie slajdy)
- zazwyczaj robimy to w dołączanym pliku nagłówkowym lub na początku kodu programu
- zakres widoczności struktur zależy od miejsca ich deklaracji tak samo jak w przypadku zmiennych — możemy tworzyć struktury zarówno globalne jak i lokalne dla funkcji
- lokalna struktura nie będzie widoczna poza funkcją, w której została zadeklarowana
- od chwili deklaracji struktury możemy używać tak jak innych typów, np. możemy tworzyć zmienne jej typu

Struktury

- w C++ możemy napisać `Data urodziny;`
- w C musimy napisać `struct Data urodziny;`
- słowo `struct` w C możemy pominąć, jeżeli skorzystamy z typedef-a:
  ```
  struct _Data { ... };
  typedef _Data Data;
  ```
  lub
  ```
  typedef struct _Data { ... } Data;
  ```
  wtedy będziemy mogli napisać `Data urodziny;`

Struktury

- np. struktura reprezentująca studenta może mieć postać:

```
typedef struct _Student {
        char imie[16];
        char nazwisko[24];
        int nrIndeksu;
        Data dataUrodzenia;
        int rokStudiow;
        } Student;
```

Struktury

- aby odwołać się do konkretnego pola struktury musimy napisać `zmienna.nazwaPola`
- np. :

```
Data urodziny;
urodziny.rok = 1990;
scanf("%d", &(urodziny.miesiac));
urodziny.dzien = urodziny.rok / 16;
printf("%d-%d-%d", urodziny.dzien,
       urodziny.miesiac, urodziny.rok);
Student s;
s.dataUrodzenia.rok = 1995;
```

Struktury — przykłady

- np. :

```
Data wydarzenia[128]; // tablica struktur
wydarzenia[1].rok = 1410;
wydarzenia[2] = wydarzenia[0];
...
int Porownaj(Data a, Data b) {
     // funkcja porównująca dwie daty
     };
Data Wczoraj() {
     // funkcja zwracająca datę
     Data wczoraj;
     ...;
     return wczoraj;
     };
```

Struktury — wejście/wyjście

- ani printf/scanf ani cin/cout nie obsługują struktur:
  nie możemy napisać `cin >> data;` czy `printf("%D", data);`
- musimy to robić samodzielnie, dla każdego pola:

```
cin >> data.dzien >> data.miesiac >> data.rok;
printf("%d-%d-%d", data.dzien, data.miesiac, data.rok);
```

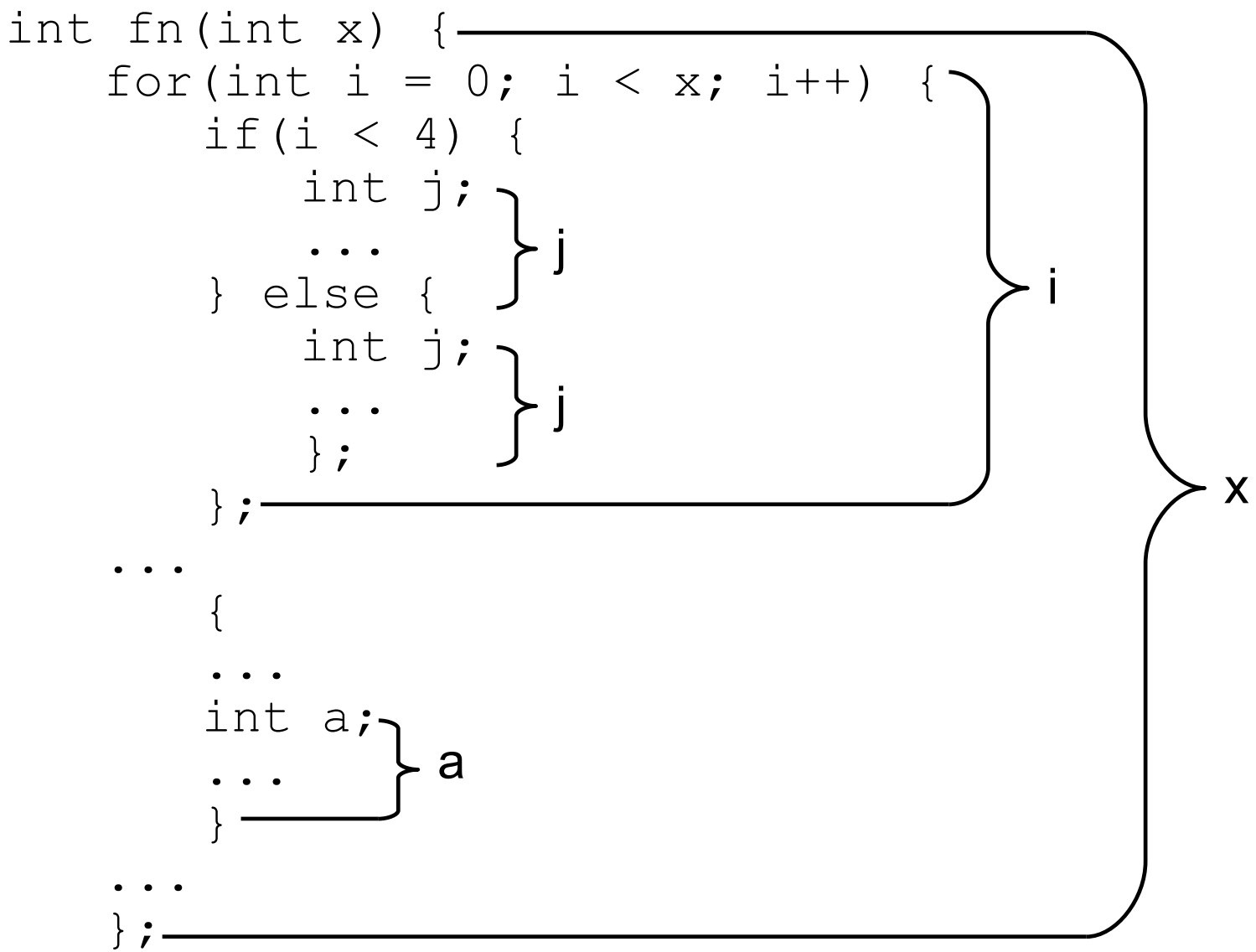
Struktury — uwagi

- w przypadku przypisania (`data1 = data2;`) struktury kopiowane są bajt po bajcie (uwaga na wskaźniki)
- struktury są przekazywane i zwracane z funkcji przez wartość

Czas życia zmiennej

- każda zmienna ma przypisany czas życia — czas od momentu powstania do jej zniszczenia,
- czas życia zmiennych globalnych to czas działania programu,
- czas życia zmiennych lokalnych obejmuje czas od jej definicji do zakończenia bloku, do którego należą (zmienne zdefiniowane w for/while należą do bloku obejmującego wnętrze pętli),
- czas życia parametrów funkcji to czas od momentu wywołania do zakończenia funkcji

Czas życia zmiennej

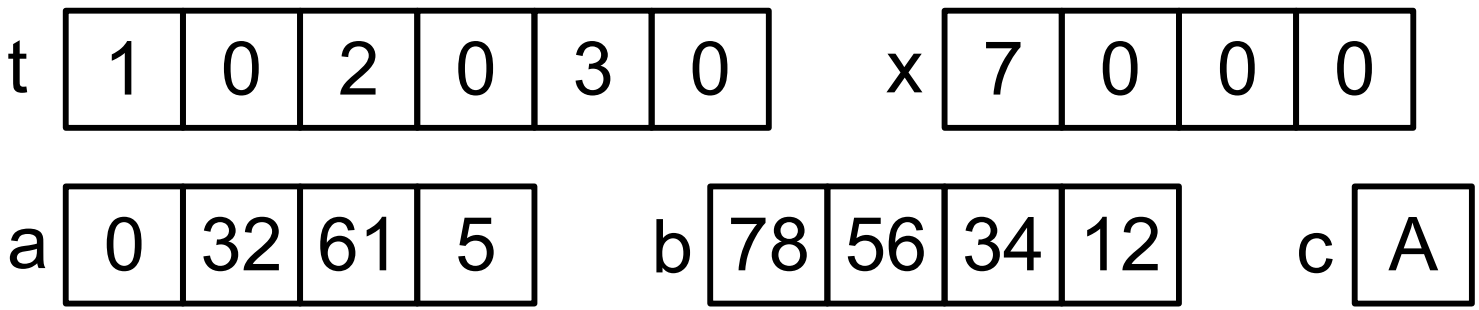


Model pamięci w C/C++

- każdy obiekt (tj. zmienna lub stała) posiada adres,
- adres ten nie zmienia się w trakcie życia zmiennej,
- różne obiekty mają różne adresy (o ile ich czasy życia nakładają się),
- obiekty zajmują pewną liczbę komórek pamięci,
- zajmowany obszar jest ciągły,
- adres = numer pierwszej komórki pamięci zajmowanej przez obiekt,
- kolejne elementy tablic zajmują kolejne grupy komórek (bez przerw)

Model pamięci w C/C++

```
int fn(int x) {
    int a;
    const int b=0x12345678;
    char c='A';
    short int t[3]={1,2,3};
    };
```

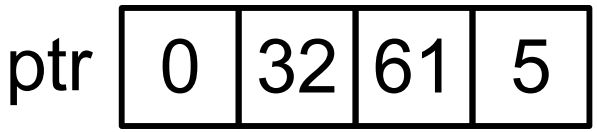


Wskaźniki

- **wskaźnik** to **zmienna** pamiętająca adres pierwszej komórki pewnego obiektu w pamięci
- **typ** wskaźnika mówi, jakiego obiektu adres pamiętamy
- np. wskaźnik na liczbę całkowitą to zmienna pamiętająca adres liczby całkowitej
- wskaźnik to tylko adres — utworzenie wskaźnika nie powoduje utworzenia wskazywanej zmiennej
- informacja o typie wskaźnika nie istnieje w trakcie wykonywania programu — typ wskaźnika jest znany wyłącznie w trakcie kompilacji
- wskaźnik na int-a o wartości np. 1000 nie mówi, że pod adresem 1000 znajduje się int, mówi jedynie, że używając tego wskaźnika chcemy, aby komórki pamięci o numerach 1000, 1001, ... były potraktowane jak int

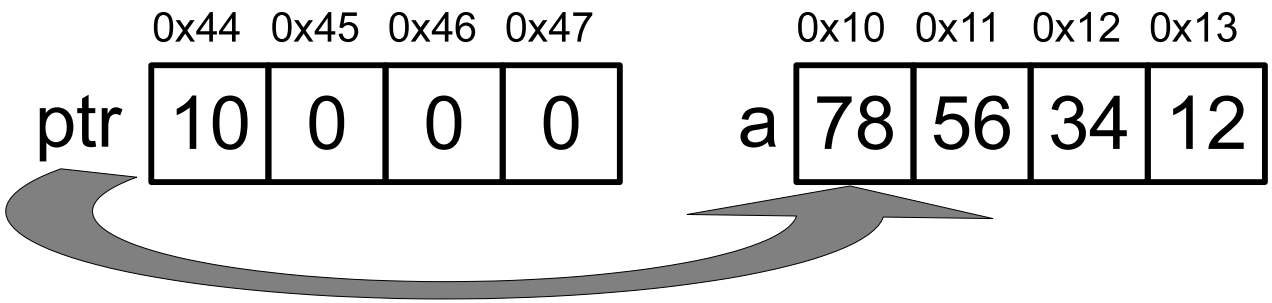
Wskaźniki

- wskaźnik deklarujemy: `Typ *nazwa;`
- np. `int *ptr;`
- to znaczy, że zadeklarowaliśmy wskaźnik o nazwie `ptr`, który będzie pokazywał na liczby typu `int`.



Pobranie adresu

- wskaźnik przechowuje adres zmiennej
- przed inicjalizacją wskaźnik pamięta pewien (nieznany) adres — tak samo jak niezainicjalizowana zmienna
- aby był użyteczny, powinien pamiętać użyteczny adres (pewnej zmiennej)
- przypisanie wskaźnikowi adresu zmiennej (zmienna powinna być takiego typu, na jaki wskazuje wskaźnik)
- `nazwa_wskaznika = &nazwa_zmiennej;`
- np. `int *ptr = &a;`



Pobranie adresu — przykład

- np.

```
int x, y;
int *p;
p = &x;
printf("p=%p &x=%p &y=%p\n", p, &x, &y);
```

```
p=0012FF88 &x=0012FF88 &y=0012FF84
```

Pobranie adresu — przykład

- np.

```
int x, y;
int *p;
p = &x;
printf("p=%p &x=%p &y=%p\n", p, &x, &y);
```

```
p=0012FF88 &x=0012FF88 &y=0012FF84
```

```
p = &y;
printf("p=%p &x=%p &y=%p\n", p, &x, &y);
```

```
p=0012FF84 &x=0012FF88 &y=0012FF84
```

Pobranie adresu — przykład

- np.

```
int x, y;
int *p;
p = &x;
printf("p=%p &x=%p &y=%p\n", p, &x, &y);
```

```
p=0012FF88 &x=0012FF88 &y=0012FF84
```

```
p = &y;
printf("p=%p &x=%p &y=%p\n", p, &x, &y);
```

```
p=0012FF84 &x=0012FF88 &y=0012FF84
```

```
printf("&p=%p\n", &p);
```

```
&p=0012FF80
```

Dereferencja

- dereferecja (wyłuskanie) — pobranie zmiennej wskazywanej przez wskaźnik: `*wskaźnik`
- dereferencja wskaźnika to wskazywana zmienna: jeżeli wskaźnik *ptr* wskazuje na zmienną *a*, to dereferencja wskaźnika *ptr* to zmienna *a*
- poprzez wskaźnik *ptr* mamy dostęp do zmiennej *a* – dereferencja to dokładnie to samo co wskazywana zmienna: wszędzie gdzie piszemy *a* możemy też napisać **ptr*
- dereferencja niezainicjalizowanego lub zawierającego błędny adres wskaźnika spowoduje (zazwyczaj) wystąpienie wyjątku programu

Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p;
p = &x;
printf("*p=%d x=%d y=%d\n", *p, x, y);
```

```
*p=1 x=1 y=2
```

Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p;
p = &x;
printf("*p=%d x=%d y=%d\n", *p, x, y);
```

```
*p=1 x=1 y=2
```

```
p = &y;
printf("*p=%d x=%d y=%d\n", *p, x, y);
```

```
*p=2 x=1 y=2
```

Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p;
p = &x;
*p = 3;
printf("*p=%d x=%d y=%d\n", *p, x, y);
```

```
*p=3 x=3 y=2
```

Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p;
p = &x;
*p = 3;
printf("*p=%d x=%d y=%d\n", *p, x, y);
```

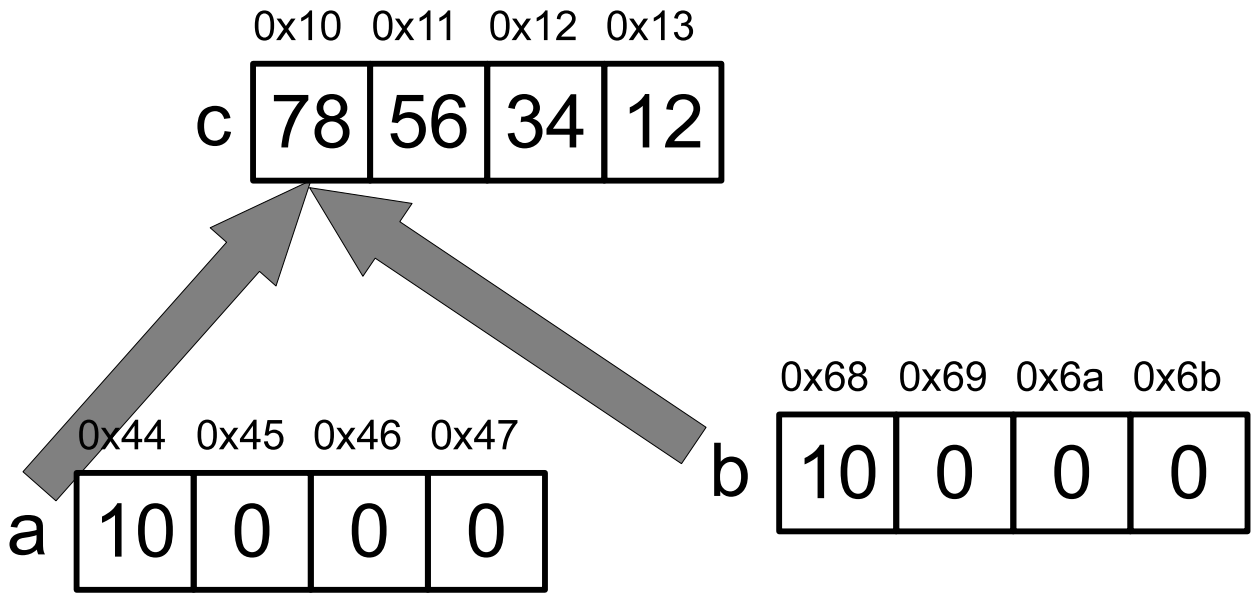
```
*p=3 x=3 y=2
```

```
x = 4;
printf("*p=%d x=%d y=%d\n", *p, x, y);
```

```
*p=4 x=4 y=2
```

Wskaźniki

- wskaźniki to normalne zmienne: można je np. kopiować
- np. a i b to wskaźniki na typ całkowity, zaś c to zmienna typu całkowitego
- jeżeli do a wpiszemy adres c a następnie wykonamy $b = a$ to zarówno a jak i b będą pokazywały na c



Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p, *q;
p = &x;
q = p;
printf("*p=%d *q=%d x=%d y=%d\n",*p,*q,x,y);
```

```
*p=1 *q=1 x=1 y=2
```

Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p, *q;
p = &x;
q = p;
printf("*p=%d *q=%d x=%d y=%d\n",*p,*q,x,y);
```

```
*p=1 *q=1 x=1 y=2
```

```
*q = 3;
printf("*p=%d *q=%d x=%d y=%d\n",*p,*q,x,y);
```

```
*p=3 *q=3 x=3 y=2
```

Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p, *q;
p = &x;
q = p;
printf("*p=%d *q=%d x=%d y=%d\n",*p,*q,x,y);
```

```
*p=1 *q=1 x=1 y=2
```

```
*q = 3;
printf("*p=%d *q=%d x=%d y=%d\n",*p,*q,x,y);
```

```
*p=3 *q=3 x=3 y=2
```

```
x = 4;
printf("*p=%d *q=%d x=%d y=%d\n",*p,*q,x,y);
```

```
*p=4 *q=4 x=4 y=2
```

Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p, *q;
p = &x;
q = &y;
printf("p=%p *p=%d q=%p *q=%d x=%d y=%d\n",
        p, *p, q, *q, x, y);
```

```
p=0012FF88 *p=1 q=0012FF84 *q=2 x=1 y=2
```

Dereferencja — przykład

- np.

```
int x = 1, y = 2;
int *p, *q;
p = &x;
q = &y;
printf("p=%p *p=%d q=%p *q=%d x=%d y=%d\n",
        p, *p, q, *q, x, y);
```

```
p=0012FF88 *p=1 q=0012FF84 *q=2 x=1 y=2
```

```
*p = *q;
printf("p=%p *p=%d q=%p *q=%d x=%d y=%d\n",
        p, *p, q, *q, x, y);
```

```
p=0012FF88 *p=2 q=0012FF84 *q=2 x=2 y=2
```

Dereferencja — przykład

- np.

```
x = 1; y = 2;
p = q;
printf("p=%p *p=%d q=%p *q=%d x=%d y=%d\n",
        p, *p, q, *q, x, y);
```

```
p=0012FF84 *p=2 q=0012FF84 *q=2 x=1 y=2
```

NULL / nullptr

- wyróżniamy pewną specjalną wartość wskaźnika (adres o wartości 0) i umawiamy się, że wskaźnik pamiętający ten adres nie pokazuje na nic
- ta specjalna wartość to NULL
- wymyślony przez Sir Charlesa Antony'ego Richarda Hoare, w roku 1965 podczas tworzenia systemu typów dla ALGOLa W

NULL / nullptr

- w 2009 roku przeprosił:
  "I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. **But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.** This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."
  [Hoare (25 VIII 2009). "Null References: The Billion Dollar Mistake"]

NULL / nullptr

- dereferencja pustego wskaźnika spowoduje wyjątek programu (NULL pointer exception, wyjątek 0xC0000005, SIGSEGV, segmentation fault)
- od C++11 istnieje również nullptr (praktycznie to samo co NULL, ale lepsze)
- jeśli możemy używamy nullptr zamiast NULL

Arytmetyka wskaźników

- do wskaźnika możemy dodać liczbę całkowitą — powoduje to przesunięcie wskaźnika o określoną liczbę obiektów do przodu
- ma to zastosowanie w przypadku tablic — jeżeli `p` pokazuje na pierwszy element tablicy, to `p+1` pokazuje na drugi, `p+2` na trzeci itd.
- analogicznie, od wskaźnika możemy odjąć liczbę całkowitą
- wskaźniki obsługują operatory: +, -, +=, -=, ++ i --
- w przypadku +, -, +=, -= drugim argumentem musi być liczba całkowita

Arytmetyka wskaźników — przykład

- np.

```
int t[3] = { 1, 2, 3 };
int *p = t; // to samo co p=&t[0]
printf("p=%p *p=%d t[0]=%d t[1]=%d t[2]=%d\n",
       p, *p, t[0], t[1], t[2]);
```

```
p=0012FF80 *p=1 t[0]=1 t[1]=2 t[2]=3
```

Arytmetyka wskaźników — przykład

- np.

```
int t[3] = { 1, 2, 3 };
int *p = t; // to samo co p=&t[0]
printf("p=%p *p=%d t[0]=%d t[1]=%d t[2]=%d\n",
       p, *p, t[0], t[1], t[2]);
```

```
p=0012FF80 *p=1 t[0]=1 t[1]=2 t[2]=3
```

```
p += 2;
printf("p=%p *p=%d t[0]=%d t[1]=%d t[2]=%d\n",
       p, *p, t[0], t[1], t[2]);
```

```
p=0012FF88 *p=3 t[0]=1 t[1]=2 t[2]=3
```

Arytmetyka wskaźników — przykład

- np.

```
int t[3] = { 1, 2, 3 };
int *p = t; // to samo co p=&t[0]
printf("p=%p *p=%d t[0]=%d t[1]=%d t[2]=%d\n",
       p, *p, t[0], t[1], t[2]);
```

```
p=0012FF80 *p=1 t[0]=1 t[1]=2 t[2]=3
```

```
p += 2;
printf("p=%p *p=%d t[0]=%d t[1]=%d t[2]=%d\n",
       p, *p, t[0], t[1], t[2]);
```

```
p=0012FF88 *p=3 t[0]=1 t[1]=2 t[2]=3
```

```
*(p-1) = 5;
printf("p=%p *p=%d t[0]=%d t[1]=%d t[2]=%d\n",
       p, *p, t[0], t[1], t[2]);
```

```
p=0012FF88 *p=3 t[0]=1 t[1]=5 t[2]=3
```

Arytmetyka wskaźników

- wskaźniki możemy ze sobą porównywać — możemy sprawdzić, czy pokazują na ten sam bajt pamięci lub sprawdzić, który pokazuje na bajt wcześniejszy
- ma to zastosowanie np. przy przeglądaniu tablic
- dwa wskaźniki tego samego typu możemy od siebie odjąć — różnicą będzie ilość obiektów dzieląca dwa wskazywane obiekty
- różnica wskaźników jest liczbą całkowitą (nie wskaźnikiem)

Arytmetyka wskaźników — przykład

- np.

```
int t[3] = { 1, 2, 3 };
int *p, *q;
p = t; // to samo co p=&t[0]
q = &t[2]; // albo q=p+2 lub q=t+2
printf("p=%p *p=%d q=%p *q=%d\n", p, *p, q, *q);
```

```
p=0012FF80 *p=1 q=0012FF88 *q=3
```

Arytmetyka wskaźników — przykład

- np.

```
int t[3] = { 1, 2, 3 };
int *p, *q;
p = t; // to samo co p=&t[0]
q = &t[2]; // albo q=p+2 lub q=t+2
printf("p=%p *p=%d q=%p *q=%d\n", p, *p, q, *q);
```

```
p=0012FF80 *p=1 q=0012FF88 *q=3
```

```
printf("q-p=%d p-q=%d\n", q-p, p-q);
```

```
q-p=2 p-q=-2
```

Arytmetyka wskaźników — przykład

- np.

```
int t[6] = { 1, 2, 3, 4, 5, 6 };
int *p, *q;
for(p=t, q=t+6; p<q; p++) {
      printf("%d ", *p);
      };
printf("\n");
```

```
1 2 3 4 5 6
```

Tablice a wskaźniki

- deklarujemy tablicę `int t[16];`
- możemy napisać `*t`, co będzie równoważne `t[0]`
- możemy napisać `*(t+1)`, co będzie równoważne `t[1]`
- możemy napisać `*(t+2)`, co będzie równoważne `t[2]`
- `t` jest w rzeczywistości wskaźnikiem na początek zadeklarowanej tablicy, ale nie możemy go modyfikować (nie jest to do końca prawda, ale na razie to nam wystarczy)
- w drugą stronę, czy wskaźniki "zachowują się" jak tablice?

Tablice a wskaźniki

- możemy napisać:

```
int t[4] = { 1, 2, 3, 4 };
int *p;
p = t;
printf("%d %d %d %d\n",
       p[0], *(t+1), *(p+2), t[3]);
```

```
1 2 3 4
```

Tablice a wskaźniki

- możemy napisać:

```
int t[4] = { 1, 2, 3, 4 };
int *p;
p = t;
printf("%d %d %d %d\n",
       p[0], *(t+1), *(p+2), t[3]);
```

```
1 2 3 4
```

```
p += 3;
printf("p-t=%d\n", p-t);
```

```
p-t=3
```

Wskaźnik na strukturę

- wskaźnik może pokazywać na dowolny typ, nawet na strukturę:

```
Data wczoraj;
Data *pDzien;
pDzien = &wczoraj;
(*pDzien).rok = 2011;
(*pDzien).miesiac = 2;
(*pDzien).dzien = 18;
```

Wskaźnik na strukturę

- zamiast `(*pDzien).pole` możemy napisać `pDzien->pole`:

```
Data wczoraj;
Data *pDzien;
pDzien = &wczoraj;
pDzien->rok = 2011;
pDzien->miesiac = 2;
pDzien->dzien = 18;
```

Wskaźnik na wskaźnik

- wskaźnik może pokazywać na dowolny typ, nawet na inny wskaźnik:

```
int x = 1;
int *pI = &x;
int **ppI = &pI;

printf("&x=%p &pI=%p &ppI=%p\n", &x, &pI, &ppI);
```

```
&x=0012FF88 &pI=0012FF84 &ppI=0012FF80
```

Wskaźnik na wskaźnik

- wskaźnik może pokazywać na dowolny typ, nawet na inny wskaźnik:

```
int x = 1;
int *pI = &x;
int **ppI = &pI;

printf("x=%d pI=%p ppI=%p\n", x, pI, ppI);
```

```
x=1 pI=0012FF88 ppI=0012FF84
```

Wskaźnik na wskaźnik

- wskaźnik może pokazywać na dowolny typ, nawet na inny wskaźnik:

```
int x = 1;
int *pI = &x;
int **ppI = &pI;

printf("&x=%p pI=%p *ppI=%p\n", &x, pI, *ppI);
```

```
&x=0012FF88 pI=0012FF88 *ppI=0012FF88
```

Wskaźnik na wskaźnik

- wskaźnik może pokazywać na dowolny typ, nawet na inny wskaźnik:

```
int x = 1;
int *pI = &x;
int **ppI = &pI;

printf("x=%d *pI=%d **ppI=%d\n", x, *pI, **ppI);
```

```
x=1 *pI=1 **ppI=1
```

Wskaźnik jako parametr funkcji

- wskaźniki pozwalają na stworzenie funkcji modyfikującej parametry
- np.:

```
void Zamien(int *x, int *y) {
      int t;
      t = *x;
      *x = *y;
      *y = t;
      };
```

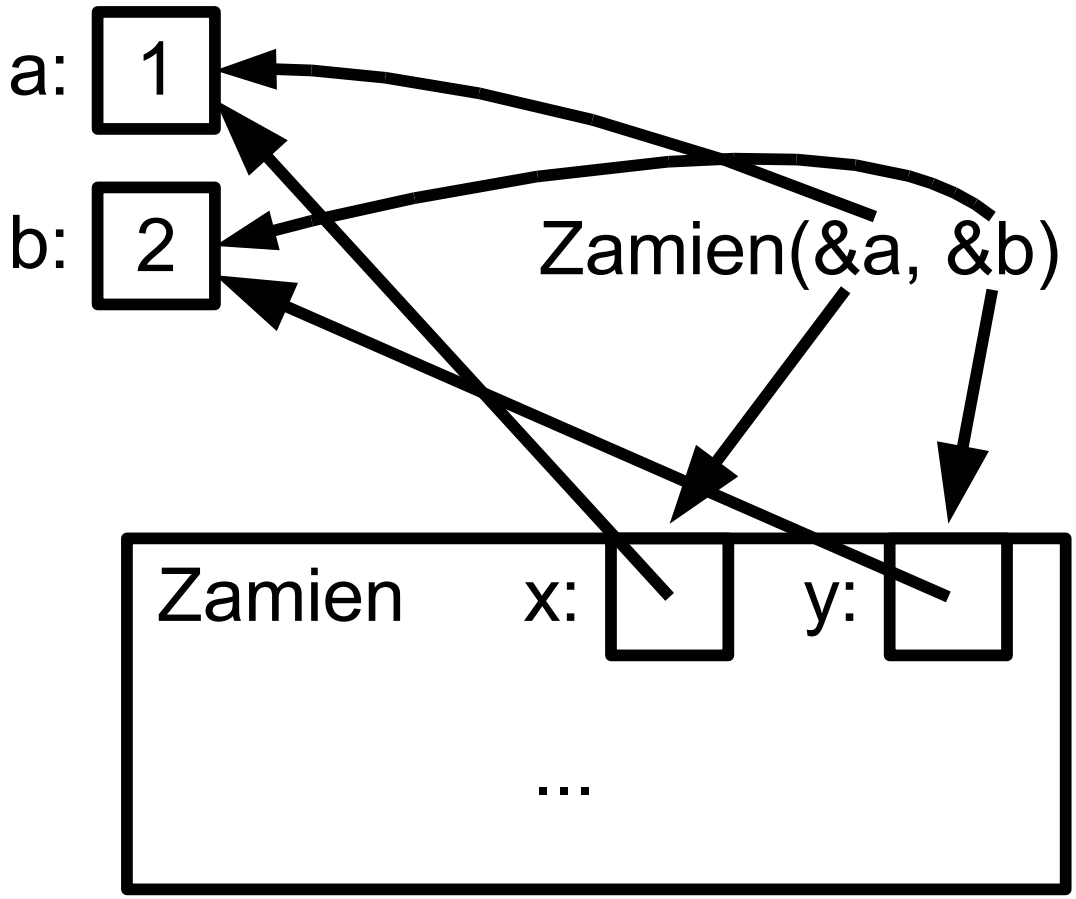
Wskaźnik jako parametr funkcji

- wołamy:

```
int a = 1, b = 2;
Zamien(&a, &b);
printf("a=%d b=%d \n", a, b);
```

```
a=2 b=1
```

Wskaźnik jako parametr funkcji



Zwracanie wskaźnika

- Funkcja może także zwracać wskaźnik, ale należy uważać:

```
int *Funkcja(int a, int b) {
      int t;
      t = a + b;
      return &t;
      };
```

- gdzie jest błąd?

Pamięć w programie

- każdy program ma do dyspozycji 4 obszary pamięci:
  - pamięć programu (zazwyczaj jest tylko odczytywana)
  - pamięć na zmienne globalne (tu przechowywane są wszystkie zmienne globalne)
  - stos (tu przechowywane są zmienne lokalne wszystkich funkcji)
  - sterta
- z punktu widzenia programu obszary te są nierozróżnialne (można jednak zaobserwować ten podział patrząc na wartości wskaźników na różne zmienne)

Pamięć w programie

- np.:

```
int g;
void fn(int a) {
      int b;
      printf("&a=%p &b=%p\n", &a, &b);
      };
int main() {
      printf("fn=%p\n", fn);
      printf("&g=%p\n", &g);
      fn();
      };
```

```
fn=00401150
&g=0040C470
&a=0012FF88 &b=0012FF7C
```

Pamięć w programie

- każdy program deklaruje ile potrzebuje:
  - pamięci programu (tyle, by się program zmieścił)
  - pamięci na zmienne globalne (suma rozmiarów wszystkich zmiennych globalnych)
  - pamięci na stos (można to ustawić w opcjach projektu lub przełącznikami kompilatora)

Sterta (heap)

- sterta to miejsce w którym system operacyjny przydziela programowi, na żądanie, nowe obszary pamięci
- sterta początkowo jest zawsze pusta, jednak może rosnąć w miarę wzrostu zapotrzebowania programu na pamięć, lub maleć, gdy pamięć przestaje być potrzebna
- *Uwaga: angielski termin heap oznacza zarówno obszar pamięci, jak i pewną strukturę danych, w języku polskim rozróżniamy: obszar pamięci nazywamy stertą a strukturę danych kopcem*

Pamięć dynamiczna

- często dopiero w trakcie działania programu dowiadujemy się, ile pamięci będziemy potrzebować
- do tego służy dynamiczny przydział pamięci — w każdej chwili możemy zażądać od systemu przyznania nam dodatkowego obszaru pamięci i jeżeli tylko w systemie jest jeszcze dostatecznie dużo wolnej pamięci, dostaniemy ją (na stercie)
- po zakończeniu korzystania z przydzielonego obszaru, należy go zwolnić
- czas życia tak utworzonych zmiennych kończy się dopiero w momencie ręcznego ich usunięcia

Przydzielenie pamięci

- aby otrzymać pamięć wystarczającą do przechowania zmiennej typu T potrzebujemy wskaźnik na typ T (niech będzie to zmienna o nazwie *ptr*)
- w C: `ptr = (T *)malloc(sizeof(T));`
- w C++: `ptr = new T;`
- aby otrzymać pamięć wystarczającą do przechowania tablicy n zmiennych typu T potrzebujemy wskaźnik na typ T (niech będzie to zmienna o nazwie *tab*)
- w C: `tab = (T *)malloc(sizeof(T) * n);`
- w C++: `ptr = new T[n];`

Zwalnianie pamięci

- aby zwolnić pamięć, potrzebujemy wskaźnik na ten obszar (wskaźnik, który otrzymaliśmy przy przydzielaniu pamięci)
- w C: `free(ptr);`
- w C++: `delete ptr;`
- w C: `free(tab);`
- w C++: `delete[] tab;`
- po zwolnieniu pamięci, zmienna wskazywana przez *ptr* (*tab*) przestaje istnieć (jej zawartość zostaje utracona)
- zawartość obszaru pamięci może zmienić się natychmiast po jego zwolnieniu, więc nie należy korzystać z zawartości wskaźnika *ptr* (*tab*)

Czas życia zmiennych dynamicznych

```
int main() {
    int *t;
    t = UtworzInty(3);

        int *UtworzInty(int ile) {
            int *tab = new int[ile];
            return tab;
            };

    ...
    UsunInty(t);

        void UsunInty(int *ptr) {
            delete[] ptr;
            };

    ...
    };
```

tab

ptr

czas życia tablicy

Czas życia zmiennych dynamicznych

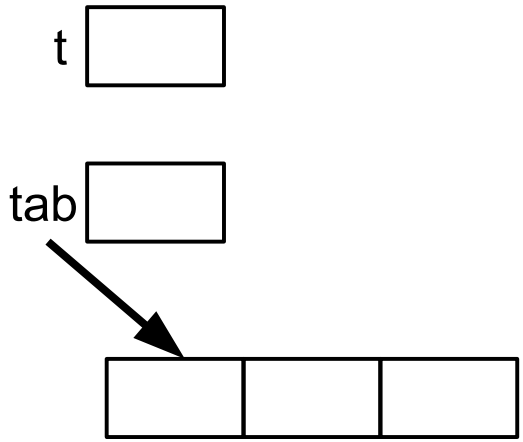
```
int main() {
    int *t;
    t = UtworzInty(3);

            int *UtworzInty(int ile) {
                int *tab = new int[ile];
                return tab;
                };

    ...
    UsunInty(t);

            void UsunInty(int *ptr) {
                delete[] ptr;
                };

    ...
    };
```



Czas życia zmiennych dynamicznych

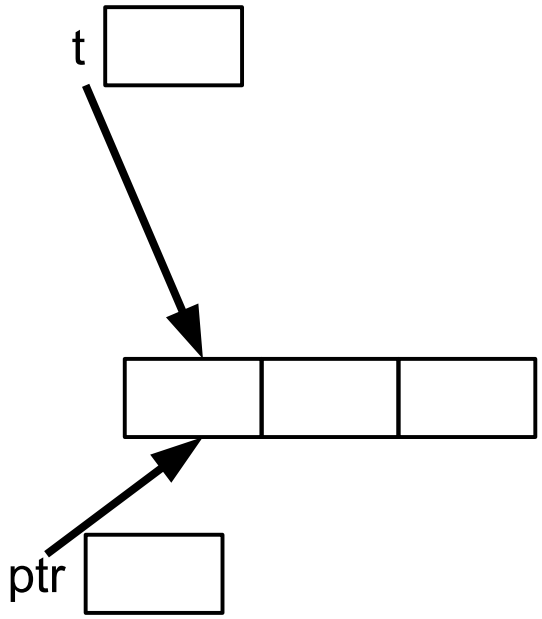
```
int main() {
    int *t;
    t = UtworzInty(3);

            int *UtworzInty(int ile) {
                int *tab = new int[ile];
                return tab;
                };

    ...
    UsunInty(t);

            void UsunInty(int *ptr) {
                delete[] ptr;
                };

    ...
    };
```



Czas życia zmiennych dynamicznych

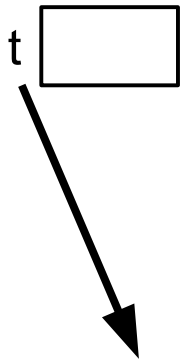
```
int main() {
    int *t;
    t = UtworzInty(3);

            int *UtworzInty(int ile) {
                int *tab = new int[ile];
                return tab;
                };

    ...
    UsunInty(t);

            void UsunInty(int *ptr) {
                delete[] ptr;
                };

    ...
    };
```



Zwalnianie pamięci

- odwołanie się do zwolnionej pamięci może (nie musi) spowodować wyjątek, ale zawsze poprowadzi do błędnego wykonania programu
- zwolnienie innego wskaźnika niż przydzielony będzie miało podobne konsekwencje
- pozostawienie obiektu w pamięci (nie zwolnienie go) powoduje wycieki pamięci — z czasem w systemie jest coraz mniej wolnej pamięci

Wyciek pamięci

- np.:

```
void fn() {
      int *p = new int[4];
      }; // tablica nie jest zwalniana
int main() {
      fn();
      // tu mamy wyciek pamięci
      };
```

Czy system robi to za nas?

- w momencie zakończenia procesu (czyli np. naszego programu) system operacyjny usuwa przydzieloną procesowi pamięć.
- to NIE oznacza, że nie musimy sprzątać po sobie

Czy system robi to za nas?

- dzieje się to tylko na koniec programu — jeżeli program działa długo i ma wycieki pamięci, po pewnym czasie zużyje całą dostępną pamięć
- system zwalnia tylko pamięć, delete dodatkowo wywołuje destruktory (w których obiekty sprzątają po sobie)
- dzięki zwalnianiu całej pamięci temu szybciej znajdziemy niektóre błędy — wyjątek wyskoczy przy zwalnianiu pamięci demaskując błąd (zakomentowane free czy delete w kodzie krzyczy: "tu jest błąd!")
- zarządzanie zasobami jest umiejętnością, którą trzeba opanować (zasoby to nie tylko pamięć ale też połączenia sieciowe, z bazą danych, wątki, pliki itp.), niezależnie od języka programowania i obecności garbage collectora

Przykład

- "Writing a very fast cache service with millions of entries in Go" (http://allegro.tech/2016/03/writing-fast-cache-service-in-go.html)

Przykład

- "Writing a very fast cache service with millions of entries in Go" (http://allegro.tech/2016/03/writing-fast-cache-service-in-go.html)
- "Why Go? [...] It also has managed memory, so it looks safer and easier to use than C/C++.[...]".

Przykład

- "Writing a very fast cache service with millions of entries in Go" (http://allegro.tech/2016/03/writing-fast-cache-service-in-go.html)
- "Why Go? [...] It also has managed memory, so it looks safer and easier to use than C/C++.[...]".
- Po czym 25% tekstu zajmuje rozdział "Omitting Garbage Collector".

Przykład

- "Writing a very fast cache service with millions of entries in Go" (http://allegro.tech/2016/03/writing-fast-cache-service-in-go.html)
- "Why Go? [...] It also has managed memory, so it looks safer and easier to use than C/C++.[...]".
- Po czym 25% tekstu zajmuje rozdział "Omitting Garbage Collector".
- Kolejne 15% to wybór serializera mimalizującego liczbę alokacji pamięci.

Zarządzanie pamięcią — właściciel zasobu

- większość problemów z zarządzaniem można rozwiązać ustalając właściciela zasobu (pamięci);
- właściciel odpowiada za zasób — tworzy go i usuwa; najczęściej utworzenie właściciela tworzy również zasób, usunięcie właściciela pociąga za sobą usunięcie zasobu;
- własność zasóbu może być przekazana, ale tylko innemu właścicielowi
- wymagane rozróżnienie wskaźników na wskaźniki–właścicieli i pozostałe
- np. notacja węgierska lub wykorzystranie systemu typów (np. `std::unique_ptr`)

Zarządzanie pamięcią — właściciel zasobu

```
int *oUtworzTablice(int n) { ... };
     // funkcja zwraca właściciela
void UsunTablice(int *oTab) { ... };
     // funkcja pobiera właściciela
int Suma(int *tab, int n) { ... };
     // funkcja pobiera wskaźnik bez prawa własności
...
int main() {
     int *oTablica = oUtworzTablice(10);
       // zwróconego właściciela zapisujemy
       // we właścicielu
      ...
```

Zarządzanie pamięcią — właściciel zasobu

```
int wyn = Suma(oTablica, 10);
   // Suma bierze wskaźnik bez prawa własności,
   // więc możemy przekazać
  ...
int *oTab2 = oTablica;
oTablica = nullptr;
   // przekazujemy własność ale nie może być dwóch
   // właścicieli więc oTablica traci
   // prawo własności
  ...
```

Zarządzanie pamięcią — właściciel zasobu

```
  ...
UsunTablice(oTab2);
oTab2 = nullptr;
   // przekazujemy własność do funkcji
   // więc oTab2 traci prawo własności
};
```

Zarządzanie pamięcią — właściciel zasobu — zasady

- `oWlasciciel2 = oWlasciciel1;`
  przekazanie własności, więc następna linia to
  `oWlasciciel1 = nullptr;`
- `wskaznik = oWlasciciel;`
  OK (o ile `wskaznik` nie żyje dłużej niż właściciel);
- `oWlasciciel = wskaznik;`
  BŁĄD; właściciel musi mieć prawo własności;
- `wskaznik2 = wskaznik;`
  OK; nie ruszamy żadnych praw;
- nigdy nie zwalniamy nie-właściciela;
- gdy kończy się czas życia właściciela, usuwamy jego własność;

Tablice wielowymiarowe

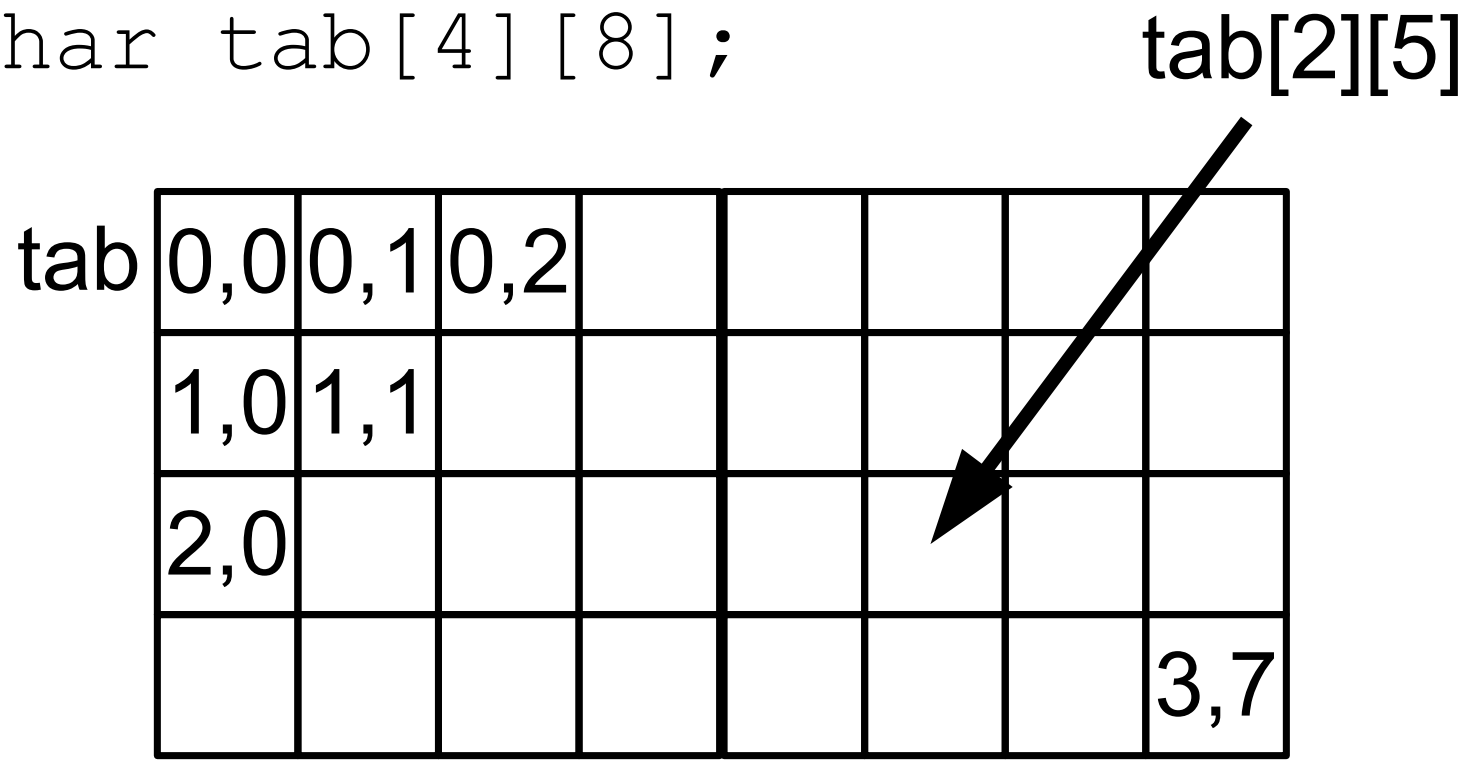
- tablice wielowymiarowe możemy tworzyć dynamicznie na dwa sposoby
- sposób 1:

```
int *tab = new int[szer * wys];
tab[x + y * szer] = 1;
...
delete[] tab;
```

- tworzymy jednowymiarową tablicę i odpowiednio numerujemy komórki (tak w rzeczywistości działają np. wielowymiarowe tablice globalne)

Tablica wielowymiarowa — sposób 1

```
char tab[4][8];
```



Tablice wielowymiarowe

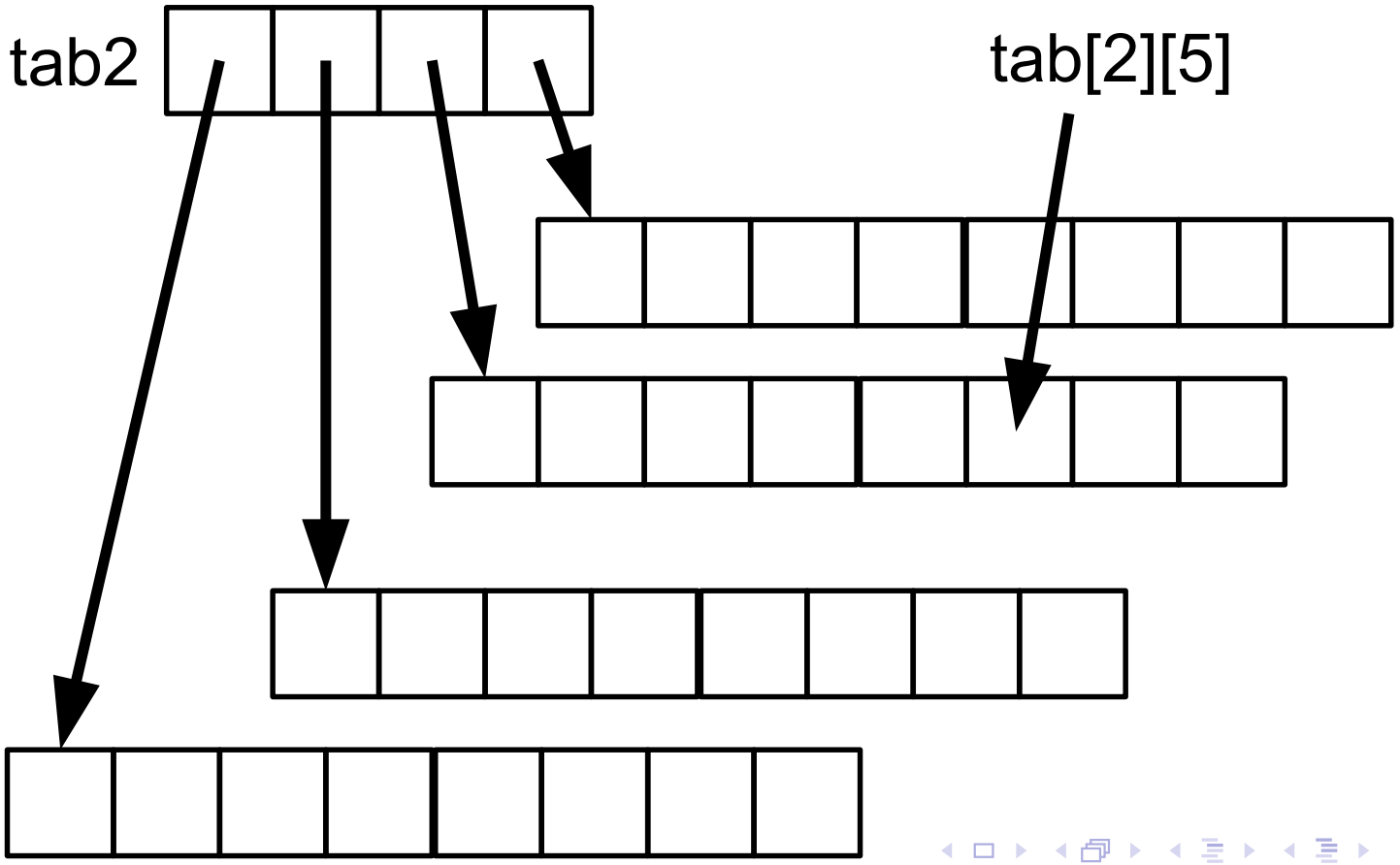
- sposób 2:

```
int **tab = new int *[wys];
for(i=0;i<wys;i++) tab[i] = new int[szer];
tab[y][x] = 1;
...
for(i=0;i<wys;i++) delete[] tab[i];
delete[] tab;
```

- tworzymy jednowymiarową tablicę tablic jednowymiarowych

Tablica wielowymiarowa — sposób 2

```
char **tab2;
tab2 = new char *[4];
for(int i = 0; i < 4; i++)
    tab2[i] = new char[8];
```



Tablice wielowymiarowe

- czym te podejścia się różnią?
- sposób 1
  - jedna alokacja i jedno zwolnienie
  - otrzymujemy ciągły obszar pamięci
  - zamiana dwóch wierszy wymaga przepisania wszystkich komórek
  - tablica ma "sztywne" wymiary

Tablice wielowymiarowe

- sposób 2
  - wiele alokacji i wiele zwolnień
  - otrzymujemy wiele rozdzielonych fragmentów pamięci
  - zamiana dwóch wierszy wymaga tylko zamiany wartości dwóch wskaźników
  - każdy wiersz może mieć inną długość, możemy zwolnić nieużywane wiersze

Wskaźniki typu `void`

- istnieje pewien specjalny typ wskaźnika — wskaźnik na void
- deklarujemy go pisząc: `void *wskaznik`
- wskaźnik taki różni się od innych tym, że nie posiada typu
- brak typu powoduje, że nie działają na nim operatory +, -, +=, -=, ++, --
- każdy wskaźnik można zamienić na wskaźnik typu void

Typowe błędy przy użyciu wskaźników

- użycie niezainicjalizowanego wskaźnika (rezultat: wyjątek C0000005)
- użycie pustego wskaźnika (NULL-a; rezultat: wyjątek C0000005)
- "wyjście" wskaźnikiem poza tablicę (rezultat: zmienne same z siebie zmieniające wartość i inne dziwne zachowania, rzadziej wyjątek C0000005; rodzaj tych "dziwnych" zachowań często podpowie nam, jakich zmiennych dotyczy problem)
- nie zwolnienie przydzielonej dynamicznie pamięci (rezultat: wyciek pamięci, utrata punktów z projektu)

Typowe błędy przy użyciu wskaźników

- dwukrotne zwolnienie wskaźnika, zwolnienie innego wskaźnika niż przydzielony (rezultat: błędy wyskakujące w najbardziej nieoczekiwanych miejscach, zazwyczaj podczas kończenia programu lub przydzielania nowej pamięci)
- wykorzystanie zwolnionej pamięci (rezultat: błędy wyskakujące w najbardziej nieoczekiwanych miejscach)

Typowe błędy przy użyciu wskaźników

- tworzenie niepotrzebnych obiektów:

```
int *wskaznik = new int;
...
wskaznik = new int[10];
```

Typowe błędy przy użyciu wskaźników

- nadużywanie pamięci dynamicznej:

```
void fn(...)  {
   int *ptr = new int;
   ...  *ptr ...
   delete ptr;
   };
```

- zamiast tego:

```
void fn(...)  {
   int zmienna;
   ...  zmienna ...
   };
```

Przydatne narzędzia: cppcheck

- narzędzie do statycznej analizy kodu
- przegląda kod źródłowy i zgłasza potencjalne problemy
- po poprawnej kompilacji STOS wyświetla raport wykonania cppchecka na kodzie
- np.:

```
[main.cpp:49]:  (error) Memory leak:  tablica
[main.cpp:176]:  (error) Uninitialized variable:  xyz
```

Przydatne narzędzia: valgrind

- narzędzie dostępne pod Linuxem
- śledzi wykonanie programu i wyłapuje problemy
- np. plik.cpp:

```
int main() {
   int *tab = new int[4];
   tab[4] = 1;
   return 0;
   };
```

- `g++ -g plik.cpp`
- `valgrind --leak-check=full ./a.out`

Przydatne narzędzia: valgrind

- wynik:

```
==28364== Memcheck, a memory error detector
...
==28364== Invalid write of size 4
==28364==    at 0x108698: main (plik.cpp:3)
==28364==  Address 0x5b7dc90 is 0 bytes after a block of size 16 alloc'd
==28364==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/...)
==28364==    by 0x10868B: main (plik.cpp:2)
...
==28364== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==28364==    at 0x4C3089F: operator new[](unsigned long) (in /usr/lib/...)
==28364==    by 0x10868B: main (plik.cpp:2)
==28364==
==28364== LEAK SUMMARY:
==28364==    definitely lost: 16 bytes in 1 blocks
...
```

Przydatne narzędzia: address sanitizer

- biblioteka współpracująca z kopilatorem, wyłapująca błędy w dostępie do pamięci
- dostępna w GCC i CLANG (taże w CLANG dla Windows)
- np. plik.cpp:

```
int main() {
   int *tab = new int[4];
   tab[4] = 1;
   return 0;
   };
```

- `g++ -g -fsanitize=address plik.cpp`

Przydatne narzędzia: address sanitizer

- rezultat:

```
=================================================================
==28100==ERROR: AddressSanitizer: heap-buffer-overflow
        on address 0xf5f007c0 at pc 0x565eb6f5 bp 0xfff55e08 sp 0xfff55df8
WRITE of size 4 at 0xf5f007c0 thread T0
    #0 0x565eb6f4 in main /test/plik.cpp:3
    #1 0xf7790e80 in __libc_start_main (/lib32/libc.so.6+0x18e80)
    #2 0x565eb580  (/test/a.out+0x580)

0xf5f007c0 is located 0 bytes to the right of 16-byte region [...)
allocated by thread T0 here:
    #0 0xf7a38244 in operator new[](unsigned int) (/usr/lib32/...)
    #1 0x565eb6b5 in main /test/plik.cpp:2
    #2 0xf7790e80 in __libc_start_main (/lib32/libc.so.6+0x18e80)

SUMMARY: AddressSanitizer: heap-buffer-overflow /test/plik.cpp:3 in main
Shadow bytes around the buggy address:
...
=>0x3ebe00f0: fa fa fa fa fa fa 00 00[fa]fa fa fa fa fa fa fa
...
```

Przydatne narzędzia: undefined behavior sanitizer

- biblioteka podobna do address sanitizera, lecz sprawdzająca niezdefiniowane zachowania
- dostępna w GCC i CLANG (taże w CLANG dla Windows)
- np. plik.cpp:

```
int main() {
   int v = -2000000000;
   v -= 1000000000;
   return 0;
   };
```

- `g++ -g -fsanitize=undefined plik.cpp`
- wynik:

```
plik.cpp:3:4: runtime error: signed
integer overflow: -2000000000 - 1000000000
cannot be represented in type 'int'
```