C++ - klasy

Konstruktor kopiujący

```
class MojaKlasa {
  char **email;
  int ile;
  public:
  MojaKlasa(int x);

};
MojaKlasa:: MojaKlasa(int x) {
  email = new char*[ile = x];
  for (int i=0;i<ile;i++) email[i] = new char[100];
}
```


C++ - klasy

Konstruktor kopiujący

```
class MojaKlasa {
  char **email;
  int ile;
  public:
  MojaKlasa(int x);
  MojaKlasa(MojaKlasa& mk);       // konstruktor kopiujący
};
MojaKlasa:: MojaKlasa(int x) {
  email = new char*[ile = x];
  for (int i=0;i<ile;i++) email[i] = new char[100];
}
MojaKlasa::MojaKlasa(MojaKlasa& mk) {
  email = mk.email;                    // czy to jest OK?
}
```


C++ - klasy

Konstruktor kopiujący

```
MojaKlasa::MojaKlasa(MojaKlasa& mk) {
  email = new char*[mk.ile];
  ile = mk.ile;
  for (int i=0;i<ile;i++) {
    email[i] = new char[100];
    strcpy(email[i],mk.email[i]);
  }
}
```


C++ - klasy

Domyślny konstruktor kopiujący

Przy tworzeniu kopii obiektów ZAWSZE używany jest konstruktor kopiujący.

Jeżeli programista nie zadeklarował dla klasy/struktury konstruktora kopiującego, kompilator zrobi go sobie sam!

Jak będzie działał?

Jeżeli składowe klasy są typu wbudowanego (**`int`**, **`float`**,..), kompilator w konstruktorze kopiującym umieści kod kopiujący „bajt po bajcie”.

Jeżeli składowe reprezentują obiekty, kompilator wywoła rekurencyjnie konstruktory kopiujące tych obiektów składowych (jeżeli obiekty składowe będą zawierały inne obiekty, to ich konstruktory kopiujące również zostaną wywołane). Jeżeli składowe-obiekty nie mają zdefiniowanych konstruktorów kopiujących, kompilator utworzy je wg tej samej reguły.

Jest to tzw. *incjalizacja za pośrednictwem elementów składowych.*

Usuwanie obiektu

DESTRUKTOR

C++ - klasy

Końcowe porządki

- Inną przyczyną wielu błędów w programach jest brak zwolnienia lub nieprawidłowe zwolnienie zmiennych dynamicznych zaalokowanych na początku lub w trakcie działania programu.
- Obiekt zawiera z reguły szereg pól, które mogą być wskaźnikami przechowującymi adresy zmiennych dynamicznych; przed usunięciem obiektu zmienne dynamiczne należy zwolnić.
- Umieszczanie instrukcji zwalniających zmienne dynamiczne w każdym miejscu, gdzie może być usuwany istniejący obiekt, mogłoby gmatwać kod programu i zwiększać znacznie jego rozmiar.
- Aby ułatwić sprzątanie dano programiście możliwość przypisania dowolnego zbioru instrukcji do czynności usuwania istniejącego obiektu. Te instrukcje wykonają się za każdym razem kiedy usuwany jest obiekt.

C++ - klasy

Przykład deklaracji:

```
class X {
  int* tab;
  public:
  ~X();          // destruktor
};
```

Po czym rozpoznajemy destruktor?

1. **nie ma nazwy** – jest tylko powtórzona nazwa klasy poprzedzona tyldą
2. **nie może zwracać żadnych wartości** – dlatego nie deklarujemy żadnego typu przed nazwą klasy

Defincja destruktora:

```
X::~X() { if (tab!=NULL) delete []tab; }
```


C++ - klasy

- Destruktor jest wywoływany zawsze podczas destrukcji obiektu przez system (destruktor obiektu nie niszczy, ale jest wykonywany tuż przed zniszczeniem)
- Obiekty utworzone jako zmienne lokalne są niszczone automatycznie w momencie opuszczenia bloku w którym zostały zadeklarowane przez sterowanie programu
- Obiekty dynamiczne utworzone poleceniem **`new`** nie są niszczone automatycznie – muszą być niszczone poleceniem **`delete`**
- Destruktor musi być bezparametrowy, co oznacza, że nie może być przeciążany (nie można tworzyć wiele destruktorów)

restrykcyjne reguły zgodności typów

TWORZENIE I USUWANIE OBIEKTÓW

C++ - tworzenie obiektów

Ogólnie mówiąc, kiedy w C++ tworzony jest obiekt, zawsze zachodzą **dwa** procesy:

1. przydzielana jest mu pamięć:
   - w obrębie obszaru danych statycznych – zanim rozpocznie się praca programu
   - na stosie – kiedy zostanie osiągnięty określony punkt realizacji programu (np. nawias klamrowy otwierający)
   - na stercie – kiedy wywołane zostanie polecenie utworzenia zmiennej dynamicznej
2. wywoływany jest konstruktor inicjalizujący tę pamięć

C++ - tworzenie obiektów

Funkcje malloc i calloc są bardzo prymitywne. Aby utworzyć na stercie instancję klasy a potem ją usunąć, należałoby napisać coś w tym rodzaju:

```
class Obj {
  …
};
int main() {
Obj *obj = (Obj*)malloc(sizeof(Obj));// alokacja pamięci
if (obj==0) {
  perror(„nie udało się zaalokować pamięci”);
  exit(EXIT_FAILURE);
}
obj->initialize();   // zamiast wywołania konstruktora
…                    // tu instrukcje naszego programu
obj->destroy();      // zamiast wywołania destruktora
free(obj);           // zwalnianie pamięci
   // TAK NIE NALEŻY TWORZYĆ OBIEKTÓW DYNAMICZNYCH!
```


C++ - tworzenie obiektów

W C++ zapominamy o **`malloc`** i **`calloc`**.

Ponieważ w C++ *wszystko* ma swój konstruktor, który musi być zawsze wywołany, używanie **`malloc`** i **`calloc`** naruszałoby tę zasadę, ponieważ one nie dają możliwości wywołania konstruktorów i destruktorów.

- Rozwiązaniem jest połączenie w jeden operator **`new`** wszystkich działań koniecznych do utworzenia obiektu.
- Podczas generowania obiektu dynamicznego poprzez wyrażenie **`new`** przydziela się na stercie niezbędną ilość pamięci i wywołuje dla niej właściwy konstruktor, np. :

```
MojTyp *MTp = new MojTyp(1,2,3);
```

C++ - tworzenie obiektów

Operator **new**

1.najpierw alokuje stosowny obszar pamięci.

2.Dopiero kiedy alokowanie zakończy się pomyślnie, przystępuje do wywołania konstruktora.

•Nie ma potrzeby sprawdzania, czy alokacja się powiodła.

•A jeżeli się nie powiodła – wywoływana jest specjalna funkcja *new handler*. Jej zadaniem jest zgłoszenie wyjątku (*o wyjątkach będzie więcej informacji w dalszej części wykładu*)

•Rezultatem działania **new** jest:

1. **zainicjalizowany obiekt**, albo
2. **obiekt-wyjątek** informujący o problemie.

C++ - tworzenie obiektów

- Wyrażeniem komplementarnym do **new** jest **delete**
- Wyrażenie **delete** najpierw wywołuje destruktor, a następnie zwalnia przydzieloną uprzednio pamięć, np. :

```
delete MTp;
```

- Operator **delete** może być wywołany wyłącznie w stosunku do obiektu utworzonego wcześniej za pomocą **new**
- Jeżeli wskaźnik usuwany za pomocą **delete** jest NULL, to nic się nie stanie.

Z tego względu niektórzy zalecają przypisanie wskaźnikowi wartości NULL zaraz po usunięciu obiektu, żeby w przypadku próby podjęcia usuwania drugi raz dla tego wskaźnika uniknąć problemów.

RESTRYKCYJNA POLITYKA KONTROLI ZGODNOŚCI TYPÓW

C++ - zgodność typów

Dlaczego w C++ mamy dużo bardziej restrykcyjną politykę kontroli zgodności typów?

W języku C funkcja **malloc** zwraca wskaźnik typu **void*** czyli po prostu adres w pamięci. Dlatego można było np. napisać:

```
struct ABC {
   …
};
struct ABC *ptr = malloc( sizeof(ABC) );
```

Akceptowana może być konwersja różnych typów adresowych występujących po lewej i prawej stronie operatora przypisania.

W C++ taka konwersja jest niedozwolona.

C++ - zgodność typów

Kontrola typów - przykład:

```
class Obj {
   …
};
Obj *ptr1 = new Obj;
void* ptr2 = ptr1;
ptr1 = new Obj;
… // tu instrukcje naszego programu
delete ptr1;
delete ptr2;
```

Problem: skąd operator **delete** ma wiedzieć jaki destruktor ma wywołać dla obiektu wskazywanego przez **ptr2** ?

C++ - zgodność typów

Kontrola typów - przykład:

Ponieważ typ **void*** nie wskazuje na żaden konkretny obiekt, operator **delete** zwolni tylko pamięć, a żadnego konkretnego destruktora nie będzie próbował wywołać.

Ewentualna konwersja do typu **void*** nie budzi zastrzeżeń – kompilator zakłada, że programista, gubiąc informację o typie obiektu, wie co robi:

```
Obj *ptr1 = new Obj;
void* ptr2 = ptr1;
```

Ale próba konwersji w druga stronę jest już traktowana w C++ jako błąd – nie tylko zgubimy wtedy informacje o prawdziwym typie obiektu, ale wprowadzimy fałszywą informację o tym, że jest innego typu:

```
ptr1 = ptr2;
```

INNE SPOSOBY INICJALIZACJI SKŁADOWYCH OBIEKTU

C++ - klasy

Inicjalizacja agregatowa zmiennej tablicowej

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

Struktury są również agregatami, dlatego:

```
struct X {
  int i;
  double f;
  char c;
};

X x1 = {1, 2.2, 'c'};
```

Ale tylko pod pewnymi warunkami..

C++ - klasy

Inicjalizacja agregatowa obiektów

jest możliwa tylko kiedy spełnione są łącznie następujące warunki:

1. klasa nie zawiera składowych `private` ani `protected`,
2. programista nie zaimplementował w niej żadnych konstruktorów,
3. klasa po niczym nie dziedziczy (nie ma swojej klasy bazowej),
4. klasa nie ma metod polimorficznych.

C++ - klasy

Tablice struktur

```
struct X {
  int i;
  double f;
  char c;
};

X x2[3] = {{1, 2.2, 'c'}, {2, 1.1, 'b'}};
```

Trzeci element tablicy zostanie zainicjowany wartością zerową dla każdego z jego pól

C++ - klasy

Przykłady inicjalizacji agregatowej:

```
struct S1 {
        int x;
        struct F1 {
                int i;
                int j;
                int a[3];
        } b;
};

union U1 {
        int a;
        const char* b;
};
```

```
S1 sa = { 1,{2,3,{4,5,6}}};   // OK
S1 sb = { 1,2,3,4,5,6};       // OK

char cr[3] = {'a'}; //  {'a', '\0', '\0'}

int ar2d1[2][2] = {{1,2},{3,4}}; //{1, 2}
                                 //{3, 4}

int ar2d2[2][2] = {1,2,3,4};     //{1, 2}
                                 //{3, 4}

int ar2d3[2][2] = {{1},{2}};     //{1, 0}
                                 //{2, 0}

U1 u1 = {1}; // OK
```


C++ - klasy

Tablice obiektów

```
class Y {
    int i;
    double f;
    char c;
public:
    Y(int ai, double af, char ac) { /* inicjalizacja */ };
};

Y y3[3]={Y(1,2.2,'c'), Y(2, 1.1, 'b'), Y(3, 3.3, 'a')};
```

Jeżeli zdefiniowany jest jawnie konstruktor, to niezależnie czy jest to struktura czy klasa i czy składowe są publiczne czy prywatne – inicjalizacja musi odbywać się za pośrednictwem konstruktora

C++ - klasy

Konstruktory typów wbudowanych

Typy wbudowane (np. **double**, **int**) różnią się od typów definiowanych przez użytkownika, ponieważ nie mają konstruktorów, a więc zmienne musza być inicjalizowane za pomocą operacji przypisania

Tak nie jest wygodnie.

Dlatego w obecnych implementacjach C++ można już pisać:

```
int a(8);
```

albo bardziej tradycyjnie (do wyboru):

```
int a = 8;
```

Jeżeli zmienna jest typu wbudowanego, lepiej pisać po staremu

C++ - klasy

Kiedy mogą przydać się konstruktory klas wbudowanych?

Np. w *liście inicjalizatorów konstruktora*

```
class X {
  int i;
  public:
  X(int a): i(a) { }
};
```

Wywołania konstruktorów umieszczone po dwukropku za nagłówkiem metody a przed otwierającym nawiasem klamrowym reprezentują listę inicjalizatorów

Po co komu taka konstrukcja?

C++ - klasy

Wewnątrz klasy można zadeklarować pole będące stałą, np.:

```
class F {
   const int rozmiar;
   public:
   F(int r);
   void fun();
};
```

Takie stałe reprezentują wartości, które są jednokrotnie inicjalizowane i nie mogą już być później zmieniane przez cały czas życia obiektu, ALE:

ICH WARTOŚĆ NIE MUSI BYĆ IDENTYCZNA WE WSZYSTKICH OBIEKTACH.

To znaczy, że musi być inicjalizowana indywidualnie dla każdego nowego obiektu.

Ale jak, skoro nie wolno pisać instrukcji zmieniających wartość stałych?

C++ - klasy

Rozwiązaniem jest napisanie instrukcji inicjalizującej **w miejscu znajdującym się poza kodem** wszelkich metod i konstruktorów. Takim miejscem jest lista inicjalizatorów konstruktora:

```
class F {
   const int rozmiar;
   public:
   F(int r): rozmiar = r {}   // tak mi nie wolno!
   F(int r): rozmiar(r) {}    // tak jest OK.
   void fun();
};

 F a(1), b(2), c(3);   // deklaracja trzech obiektów
```


C++ - klasy

Inicjalizacja składowych zadeklarowanych na podstawie innych klas

Definiując klasę, możemy deklarować jej składowe zarówno na podstawie typów wbudowanych jak i klas

```
class X;
class Y {
    public:
      int a,b;   // składowe typu wbudowanego int
      X c;       // składowa typu takiego jak klasa
      Y();       // konstruktor domyślny
};
```

Inicjalizacja dla składowych typów wbudowanych (**a** i **b**) polega po prostu na utworzeniu zmiennej, czyli przydzieleniu pamięci.

A co ze składowymi zadeklarowanymi na podstawie klas?

C++ - klasy

Składowe zdefiniowane na podstawie klas mają również przydzielaną pamięć, ale potem następuje jeszcze wywoływanie konstruktora domyślnego..

Koniecznie domyślnego?

C++ - klasy

Składowe zdefiniowane na podstawie klas mają również przydzielaną pamięć, ale potem następuje jeszcze wywoływanie konstruktora domyślnego..

.. chyba, że programista dla tych składowych sam wskaże, który konstruktor ma być uruchomiony umieszczając jego wywołanie w liście inicjalizatorów konstruktora:

```
class X {
public:
    X();
    X(double z);
};
class Y {
public:
    int a,b;       // składowe typu wbudowanego int
    X c;           // składowa typu abstrakcyjnego
    Y(): c(0) { a= 0; b = 0; }
};
```


C++ - klasy

Pola statyczne w klasach:

```
Plik h:
class F {
    const int rozmiar;
    static int MAX_ROZMIAR;
public:
    F(int r): rozmiar(r) {}
    void fun() {}
};
Plik cpp:
int F::MAX_ROZMIAR = 100;
```

Pole statyczne deklaruje się za pomocą słowa **`static`**

Pole statyczne jest wspólne dla wszystkich instancji (!)

C++ - klasy

Pola statyczne stałe, o wartościach określonych podczas kompilacji, w klasach:

```
class F {
    const int rozmiar;
    static const int MAX_ROZMIAR = 100;
public:
    F(int r): rozmiar(r) {}
    void fun() {}
};
```

Pole statyczne stałe deklaruje się za pomocą słowa **`static const`**

Pole statyczne stałe jest wspólne dla wszystkich instancji (!)

Pole statyczne stałe inicjalizuje się w miejscu jego deklaracji (ale tylko pola typu *integral*!)

C++ - klasy

Pola statyczne stałe, o wartościach określonych podczas kompilacji, w klasach:

```
Plik h:
class F {
    const int rozmiar;
    static const double 2PI;
public:
    F(int r): rozmiar(r) {}
    void fun() {}
};

Plik cpp:
const double F::2PI = 6.283185335194;
```


C++ - klasy

Pola referencyjne w klasach:

```
Plik h:
class G;
class F {
    const int rozmiar;
    G& g;
public:
    F(int r, G& tempg): rozmiar(r), g(tempg) {}
    void fun() {}
};
```

Dla pól referencyjnych (tak jak dla zmiennych referencyjnych) można wskazać do jakiej zmiennej/obiektu mają by odniesieniem poprzez inicjalizację ale nie można tego zrobić poprzez operację przypisania.

C++ - klasy

Pola referencyjne w klasach:

```
class G;
class F {
    G& g1;
    G g2;
public:
    F(G& tempg1, G& tempg2): g1(tempg1) {
        g2 = tempg2;
    }
};
```

W przykładzie powyżej: pole **`g2`** najpierw zostanie utworzone i zainicjalizowane konstruktorem domyślnym (ponieważ wszystkie pola niewymienione w liście inicjalizatorów, są incjalizowane konstruktorami domyślnymi) a następnie operator przypisania skopiuje do niego zawartość **`tempg2`**.

C++ - klasy

Pola referencyjne w klasach:

```
class G;
class F {
    G& g1;
    G g2;
public:
    F(G& tempg1, G& tempg2): g1(tempg1) {
          g2 = tempg2;
    }
};
```

Uwaga: pole **g1** musi być inicjalizowane w liście inicjalizatorów, ponieważ nie ma konstruktora domyślnego dla zmiennych referencyjnych. Dlatego pominięcie tej inicjalizacji powoduje błąd kompilacji.

147

C++ - klasy

Pola referencyjne w klasach:

```
class G;
class F {
    G& g1;
    G g2;
public:
    F(G& tempg1, G& tempg2): g1(tempg1), g2(tempg2) {}
};
```

Tak jest prościej oraz mamy mniejszy koszt obliczeniowy: pole **g2** jest od razu inicjalizowane obiektem **tempg2**.

Komentarz: pola referencyjne w klasach to nie jest dobry pomysł.

148

C++ - klasy

Niejasności wynikające z nowych reguł C++11:

```
class X {
    int a = 1234;       // to jest składnia z C++11
public:
    X() = default;
    X(int z) : a(z) {}
};
```

Jaka wartość zostanie przypisana do ‚a' (uwaga: mamy dwa inicjalizatory dla jednego pola)?

O nowych możliwościach, które wprowadza C++11, będzie mowa w dalszej części wykładu; obecnie należy ograniczać się do składni C++98.

149