

## INNE SPOSOBY INICJALIZACJI SKŁADOWYCH OBIEKTU

© UKSW, WMP, SNS, Warszawa 131

131

## 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..*

© UKSW, WMP, SNS, Warszawa 132

132

## 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.

© UKSW, WMP, SNS, Warszawa 133

133

## 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

© UKSW, WMP, SNS, Warszawa 134

134

## C++ - klasy

### Przykłady inicjalizacji agregatowej:

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

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

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

© UKSW, WMP, SNS, Warszawa 135

135

## 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

© UKSW, WMP, SNS, Warszawa 136

136

## C++ - klasy

### Konstruktory typów wbudowanych

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

*Tak nie jest wygodnie.*

Dlatego w obecnych implementacjach C++ można już pisać:

```
int a(8);  
albo (do wyboru):  
int a = 8;
```

*Jeżeli zmienna jest typu wbudowanego, lepiej pisać po staremu*

© UKSW, WMP, SNS, Warszawa

137

137

## 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?

© UKSW, WMP, SNS, Warszawa

138

138

## 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?*

© UKSW, WMP, SNS, Warszawa

139

139

## 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
```

© UKSW, WMP, SNS, Warszawa

140

140

## 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?*

© UKSW, WMP, SNS, Warszawa

141

141

## C++ - klasy

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

*Koniecznym domyślnego?*

© UKSW, WMP, SNS, Warszawa

142

142

## C++ - klasy

Składowe zdefiniowane na podstawie klas mają również przydzielaną pamięć, ale potem następuje jeszcze wywołanie 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; }
};
```

© UKSW, WMP, SNS, Warszawa

143

143

## 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 (!)
```

© UKSW, WMP, SNS, Warszawa

144

144

## 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 integrali!](#))

© UKSW, WMP, SNS, Warszawa

145

145

## 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;
```

© UKSW, WMP, SNS, Warszawa

146

146

## 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 być odniesieniem poprzez inicjalizację ale nie można tego zrobić poprzez operacje przypisania.

© UKSW, WMP, SNS, Warszawa

147

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;
    }
};
```

Pole **g2** najpierw zostanie utworzone i zainicjalizowane konstruktorem domyślnym (ponieważ wszystkie pola niewymienione w liście inicjalizatorów, są inicjalizowane konstruktorami domyślnymi) a następnie operator przypisania skopiuje do niego zawartość **tempg2**.

© UKSW, WMP, SNS, Warszawa

148

148

## 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.

© UKSW, WMP, SNS, Warszawa

149

149

## 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.*

© UKSW, WMP, SNS, Warszawa

150

150

## 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”?

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

© UKSW, WMP, SNS, Warszawa

151

151

## C++ - klasy

### Metody stałe w klasie

Zadeklarowanie metody jako stałej stanowi „obietnicę”, że wykonanie tej metody nie zmieni stanu obiektu, na rzecz którego została wykonana. Umieszczamy **const** między nawiasem zamykającym listę argumentów a średnikiem lub nawiasem klamrowym otwierającym:

```
class X {
    int i;
public:
    X(int a=0);
    void funconst(X &ox) const;
    void fun(X &ox);
};
```

© UKSW, WMP, SNS, Warszawa

152

152

## C++ - klasy

Jeżeli zadeklarujemy obiekt jako stały, to na jego rzecz możemy wywoływać tylko metody stałe:

```
X x1(0), x2(1);
const X x3(2);

x1.funconst(x2); // OK.
x1.fun(x2); // OK.
x1.funconst(x3); // OK.
x1.fun(x3); // Źle!
x3.fun(x1); // Źle!
x3.funconst(x1); // OK.
```

© UKSW, WMP, SNS, Warszawa

153

153

## C++ - klasy

### Argumenty wywołania metody/funkcji zadeklarowane jako stałe

Jeżeli wybrane argumenty zostaną zadeklarowane jako stałe, to nie wolno zmieniać ich wartości w kodzie funkcji. Jest to gwarancja bezpieczeństwa danych dla użytkownika tej metody/funkcji.

Np. funkcja do kopiowania łańcuchów tekstowych:

```
char *strcpy(char* strTo, const char* strFrom);
```

Funkcja przyjmuje dwa argumenty: skąd kopiować i dokąd kopiować.

W nagłówku jest zadeklarowane, że obszar pamięci wskazany przez **strFrom** nie zostanie zmodyfikowany w wyniku działania tej funkcji.

© UKSW, WMP, SNS, Warszawa

154

154

## C++ - klasy

### Konstruktor kopiujący (*dokończenie*)

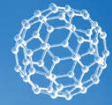
Konstruktor kopiujący nie powinien modyfikować obiektu, który jest mu podawany w argumencie wywołania, dlatego można zadeklarować ten argument jako stały (nie podlegający zmianom):

```
class MojaKlasa {
public:
    MojaKlasa(); // domyślny
    MojaKlasa(const MojaKlasa& mk); // kopiujący
    ...
};
```

© UKSW, WMP, SNS, Warszawa

155

155



## PRZECIĄŻANIE METOD

© UKSW, WMP, SNS, Warszawa

156

156

## C++ - klasy

### Przeciążanie nazw metod:

Intuicyjne dla człowieka jest nadawanie jednakowych nazw czynnościom, które mają jakąś jedną wspólną istotną właściwość (jedną lub więcej), aczkolwiek mogą różnić się w sposobie wykonania

Przykład: *mycie*

Czynność *mycie rąk* różni się od czynności *mycie szyb*, jednak łączą te dwie czynności rezultat – przywrócenie czystości

Tworząc obiekty na podstawie pojęć, może zdarzyć, że do pojęcia przypisujemy usługę, która różni się w działaniu w zależności od tego, dla kogo jest realizowana

Przykład: metoda, która wypisuje słownie liczbę. Inny tekst powinien pojawić się dla liczb całkowitych, a inny dla rzeczywistych

© UKSW, WMP, SNS, Warszawa

157

157

## C++ - klasy

### Przeciążanie nazw metod:

Przykład: metoda, która wypisuje słownie liczbę:

- Jeżeli liczba całkowita 12, napis: **dwanaście**
- Jeżeli liczba rzeczywista 12, napis: **dwanaście przecinek zero**

Możemy w klasie zadeklarować dwie metody o tej samej nazwie **napisz\_slownie**, różniące się argumentami wywołania:

```
class X {
public:
    void napisz_slownie(int);
    void napisz_slownie(double);
};
```

© UKSW, WMP, SNS, Warszawa

158

158

## C++ - klasy

### Przeciążanie nazw metod:

Tak samo jak metody, przeciążamy konstruktory:

```
class X {
    int i;
public:
    X(); // konstruktor domyślny
    X(int a); // inny konstruktor
};
```

Możemy też redukować liczbę metod/konstruktorów, stosując argumenty domyślne, np.:

```
...
X(int a=0);
```

© UKSW, WMP, SNS, Warszawa

159

159

## C++ - klasy

### Argumenty domyślne:

```
class X {
    int i;
public:
    X(int a=0);
    void fun(int, int, int a=0, double b = 0, double c = 0);
};
```

```
X x1[3] = {X(), X(1), X(2)};
```

Pierwszy z obiektów w tablicy będzie inicjalizowany konstruktorem, w którym przyjęto domyślną wartość argumentu – 0.

Zasada: domyślne mogą być tylko końcowe argumenty (może być ich kilka).

```
x1.fun(12, 21, 1); // pominięto argumenty b i c
// - przyjmą wartości 0
```

© UKSW, WMP, SNS, Warszawa

160

160



## LISTY DYNAMICZNE

© UKSW, WMP, SNS, Warszawa 161

161

## C++ - listy dynamiczne

### Listy jednokierunkowe obiektów

Z dynamicznych obiektów możemy tworzyć listy dynamiczne w taki sam sposób, jak z dynamicznych zmiennych strukturalnych

<pre>struct film_t {     char tytul[80];     int rok;     struct film_t *nast; };</pre>	<pre>class film_t { public:     char tytul[80];     int rok;     film_t *nast; };</pre>
---	---

© UKSW, WMP, SNS, Warszawa 162

162

## C++ - listy dynamiczne

Wykorzystanie konstruktora może uprościć kod:

<pre>class film_t { public:     char tytul[80];     int rok;     film_t *nast;     film_t() { tytul[0]=0;               rok=0;               nast=NULL; };     film_t(char*s, int r) {         strcpy(tytul, s);         rok=r;         nast=NULL;     }; };</pre>	<pre>FILE*stream; film_t *glowa, *wsk; ... // otwarcie pliku int r; fscanf(stream,"%s %i\n",buffer,r); while (!feof( stream )) {     if (glowa == NULL)         glowa=wsk=new film_t(buffer,r);     else {         wsk-&gt;nast=new film_t(buffer,r);         wsk=wsk-&gt;nast;     }     fscanf(stream,"%s %i\n",buffer,r); } ... // dalszy kod programu</pre>
--	---

© UKSW, WMP, SNS, Warszawa 163

163

## C++ - listy dynamiczne

Niektóre czynności można zamknąć w kodzie destruktora:

<p>Usuwanie tradycyjne (tak jak dla struktur):</p> <pre>wsk = glowa; while (glowa != NULL) {     glowa = glowa-&gt;nast;     delete wsk;     wsk = glowa; };</pre>	<p>Usuwanie z wykorzystaniem destruktora:</p> <pre>film_t::~film_t() {     delete nast; }; ... delete glowa; // rekurencja!</pre>
--	---

© UKSW, WMP, SNS, Warszawa 164

164

## C++ - listy dynamiczne

### znajdowanie elementu w liście


(przyjmijmy, że polem kluczowym w liście jest rok – dla każdego roku jest tylko jeden element):  
Szukamy np. dla: rok == 2007

Wersja dla struktur:

<pre>wsk = glowa; if (wsk-&gt;rok!=2007) {     while ((wsk-&gt;nast != NULL) &amp;&amp;            (wsk-&gt;nast-&gt;rok != 2007))         wsk = wsk-&gt;nast; } ... wsk = glowa-&gt;szukaj(2007);</pre>	<p>Szukanie z wykorzystaniem metody:</p> <pre>film_t *film_t::szukaj(int r) {     pokaz();     if (rok == r)         return this;     else         if (nast != NULL)             return nast-&gt;szukaj(r);         else             return NULL; }; ... wsk = glowa-&gt;szukaj(2007);</pre>
--	--

© UKSW, WMP, SNS, Warszawa 165

165



## this

© UKSW, WMP, SNS, Warszawa 166

166

## C++ - klasy

### Wskaźnik `this`

Kiedy wywołujemy metodę na rzecz istniejącego obiektu, to możemy w tej metodzie wywołać publiczne metody innych obiektów, np.:

```
class Pierwsza {
    void fun1(Druga*);
};
void Pierwsza::fun1(Druga *d) {
    d->j = 0;
    ... // tutaj pozostałe instrukcje fun1
}
```

Przyjmijmy, że w klasie `Druga` jest metoda `fun2`, której argumentem jest wskaźnik do obiektu klasy `Pierwsza` i chcemy właśnie tę metodę wywołać w metodzie `fun1`:

© UKSW, WMP, SNS, Warszawa

167

167

## C++ - klasy

### Wskaźnik `this`

```
void Druga::fun2(Pierwsza *p) {
    ...
}
void Pierwsza::fun1(Druga *d) {
    d->j = 0;
    d->fun2( co tu napisać? );
}
```

Problem: metoda `fun1` należąca do klasy `Pierwsza` chce wywołać metodę `fun2` należącą do klasy `Druga`. Argumentem wywołania `fun2` jest wskaźnik na obiekt typu `Pierwsza`. To wywołanie `fun2` odbywa się wewnątrz metody należącej do `Pierwsza`, więc w wywołaniu należy przekazać wskazanie na samego siebie. *Tylko skąd je wziąć?*

© UKSW, WMP, SNS, Warszawa

168

168

## C++ - klasy

### Wskaźnik `this`

..należy odwołać się do wskaźnika `this`, który zawiera adres pożądanego obiektu.

Każda z metod klasy, niezadeklarowanych jako `static`, w momencie wywołania otrzymuje, oprócz argumentów z jawnie zadeklarowanej listy parametrów wywołania, wskazanie na obiekt, na rzecz którego wystąpiło wywołanie. To wskaźnik o nazwie `this`.

Kompilator jest odpowiedzialny za inicjalizację `this` prawidłowym adresem obiektu.

Wskaźnika `this` nie można modyfikować, np. przypisać mu innej wartości.

© UKSW, WMP, SNS, Warszawa

169

169

## C++ - klasy

Teraz już wiadomo co wpisać:

```
void Druga::fun2(Pierwsza *p) {
    ...
}
void Pierwsza::fun1(Druga *d) {
    d->j = 0;
    d->fun2( this );
}
```

W ten sposób w kodzie należącym do obiektu możemy manipulować na tymże obiekcie w taki sam sposób, jakbyśmy działali na dowolnym innym tego samego typu.

© UKSW, WMP, SNS, Warszawa

170

170

## C++ - klasy

### Kiedy jeszcze używać `this`:

```
class MyClass
{
public:
    void f(int);
private:
    int x;
};

void MyClass::f(int x)
{
    ..
    this->x = x;
    ..
}
```

© UKSW, WMP, SNS, Warszawa

171

### Ale można tego uniknąć.. Np. tak:

```
class MyClass
{
public:
    void f(int);
private:
    int x;
};

void MyClass::f(int xval)
{
    ..
    x = xval;
    ..
}
```

171