

Wykład 2

Klasy i obiekty

dr Marcin Denkowski

Lublin, 2019

AGENDA

1. Referencja a wskaźnik
2. Dynamiczna alokacja pamięci
3. Konstruktory i destruktory
4. Kopiowanie obiektów
5. Składowe statyczne i stałe

PARAMETRY DOMYŚLNE FUNKCJI

- *Default function parameters*
- Parametr domyślny , wartość używana gdy nie zostanie podana jawnie wartość parametru
- Wartość parametru domyślnego jest podawana w deklaracji funkcji

```
char* left(    const char* str, int n = 1);  
char* within( const char* str, int n = 1, int j=5);  
char* without(const char* str, int n = 1, int j); // ZLE  
  
int main()  
{  
    char* str = „Ala ma kota”;  
    left(str);      // left(str, 1);  
    left(str, 5);  
    within(str, 3); // within(str, 3, 5)  
    within(str, ,10); // ZLE  
}
```

REFERENCJA

- Zmienna referencyjna – typ złożony, alias na istniejącą zmienną

```
int measure = 40;  
int& pomiar = measure; // referencja, typ: int&  
  
std::cout << "measure address: " << &measure << endl;  
std::cout << "pomiar address: " << &pomiar << endl;
```

- Ma identyczny adres jak zmienna, na którą wskazuje
- Jest tylko inną nazwą na tą samą zmienną
- Musi być zainicjalizowana podczas deklaracji
- Nie ma możliwości „przestawienia” jej na inną zmienną

REFERENCJA JAKO ARGUMENT FUNKCJI

- Zmienna referencyjna w argumencie funkcji

```
void swap_v(int a, int b) {int _a=a; a=b; b=_a;}
void swap_p(int* a, int *b) {int _a=*a; *a=*b; *b=_a;}
void swap_r(int& a, int& b) {int _a=a; a=b; b=_a;}

int main()
{
    int a = 12, b = 7;

    swap_v(a, b);      // za pomoca wartosci; nie powiedzie sie
    swap_p(&a, &b);    // OK, za pomoca wskaznikow
    swap_r(a, b);      // OK, za pomoca zmiennych (referencji)

    return 0;
}
```

- Kiedy jej używać?

WSKAŹNIK **this**

- Wskaźnik **this** – p-wartość, wskaźnik, którego adresem jest sam obiekt, na rzecz którego wywoływana jest dana metoda
- Może być używany:
 - wewnątrz nie-statycznych metod
 - wewnątrz domyślnej inicjalizacji składowych

```
class Typ{  
    float var;  
  
public:  
    void setVar(float var) //parametr var przykrywa nazwe Typ::var  
    {  
        this->var = var;  
    }  
  
    Typ* self() { return this; } // metoda zwraca adres obiektu, na rzecz  
                                //którego została wywołana  
};
```

OPERATORY NEW I DELETE

- **Operator new** – alokacja pamięci sterty (odpowiednik funkcji `malloc()`)
- **Operator delete** – dealokacja pamięci sterty (odpowiednik funkcji `free()`)
- Mają dwie wersje:
 - (1) pojedynczy obiekt
 - (2) tablica obiektów

```
(1) int* value = new int;  
Node* n = new Node();  
delete value;  
delete n;
```

```
(2) int* measure = new int[10];  
Node* pn = new Node[4];  
delete[] measure;  
delete[] pn;
```

- Nie są to zastępniki funkcji `malloc()` i `free()` i nie można ich używać wymiennie
- W obu przypadkach wykonują pewną dodatkową funkcjonalność
- Oba operatory można przeciążać

KONSTRUKTOR

- **Konstruktor** – specjalna nie-statyczna metoda klasy, która jest wywoływana zawsze, gdy następuje utworzenie nowego obiektu klasy (jawnie i niejawnie)
- Musi mieć nazwę identyczną z nazwą klasy
- Może ich być dowolna ilość (przeciążanie)
- Nie posiada typu zwracanego
- Główne zadanie to inicjalizacja składowych

```
class Vector3d {  
    float x,y,z;  
  
public:  
    Vector3d() { //konstruktor domyslny  
        x=y=z= 0.0f;  
    }  
  
    Vector3d(float x, float y, float z) { //konstruktor inicjujacy  
        this->x = x;  
        this->y = y;  
        this->z = z;  
    }  
    ...  
};
```


KONSTRUKCJA OBIEKTÓW

- Sposoby konstrukcji obiektów:

```
class Vector3d;

int main()
{
    Vector3d vec1;
    Vector3d vec2 = Vector3d();
    Vector3d vec3{5,6,7};

    Vector3d vec4(1,2,3);
    Vector3d vec5 = Vector3d(2,3,4);

    Vector3d* pvec1 = new Vector;
    Vector3d* pvec2 = new Vector();
    Vector3d* pvec3 = new Vector(5,6,7);

    Vector3d vec6(); // UWAGA: deklaracja funkcji
}
```

KONSTRUKTOR KOPIUJĄCY

- Konstruktor kopiujący klasy **T** to konstruktor, który przyjmuje w argumencie referencję Na **T&** lub **const T&**

```
class T{  
public:  
    T(const T&); // konstruktor kopiujący  
    ...  
};
```

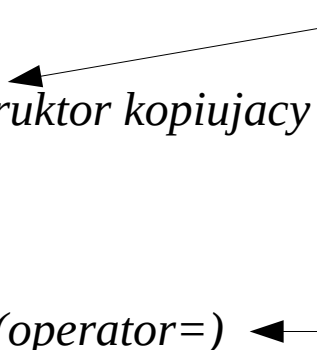
- Konstruktor kopiujący jest wywoływany zawsze gdy obiekt jest inicjowany innym obiektem tego samego typu

```
T object;  
T obj2(object);           // jawne wywołanie konstruktora kopiującego  
T obj3 = object;          // niejawne wywołanie (nie jest to przypisanie!!)
```

- Jeżeli nie ma jawnej definicji konstruktora kopiującego, kompilator zawsze utworzy jego domyślną, publiczną, trywialną wersję inline

KONSTRUKTOR KOPIUJĄCY A PRZYPISANIE

```
class Typ;  
  
Typ variable;  
Typ a = variable;  //konstruktor kopiujacy  
  
Typ b;  
...  
b = a;  //operacja przypisania (operator=)
```



Konstruktor kopiujący zostanie użyty tylko podczas inicjalizacji nowego obiektu

Operator przypisania zostanie użyty wtedy, gdy wartość jednego obiektu ma zostać przypisana innemu

KONSTRUKTOR KONWERTUJĄCY

- Konstruktor konwertujący (rzutujący) (*converting constructor*) jest używany podczas inicjalizacji kopiującej

```
class T{  
public:  
    T(int i); // konstruktor kopiujacy  
    ...  
};  
  
int main(){  
    T t1 = 1;      // inicjalizacja kopiujaca T::T(int)  
    T t2(2);       // inicjalizacja bezposrednia T::T(int)  
    T t3 = (T)6;   // inicjalizacja z rzutowaniem T::T(int)  
}
```

KONSTRUKTOR DELEGUJĄCY

- Konstruktor delegujący (*delegating constructor*) to konstruktor wywołujący inny konstruktor za pomocą listy inicjalizacyjnej konstruktora

```
class Typ {  
public:  
    Typ(float a, int n) {}  
    Typ(int n) : Typ(42, n) {}  
    ...  
};
```

METODY STATYCZNE

- (*static member function*)
- Mogą być wywoływane na rzecz samej klasy
- Mają dostęp do statycznych składowych klasy, innych metod statycznych oraz pozostałych funkcji spoza klasy

```
class Typ {  
public:  
    static int n;  
    static void func();  
    ...  
};  
int Typ::n = 3;  
  
void Typ::func()  
{  
    cout << „Metoda statyczna func() klasy Typ: ” << n << endl;  
}  
  
int main()  
{  
    Typ::func();  
    Typ t;  
    t.func();  
}
```

DESTRUKTOR

- **Destruktor** – specjalna metoda klasy, która jest wywoływana zawsze, przed dealokacją/usunięciem obiektu
- Ma nazwę identyczną z nazwą klasy poprzedzoną symbolem ~ (tylda)
- Nie posiada typu zwracanego

```
class ArrayInt {  
    int* m_array;  
    int m_size;  
  
public:  
    ArrayInt(int n) {  
        m_array = new int[n]; m_size = n;  
    }  
  
    ~ArrayInt() {  
        delete[] m_array;  
    }  
};
```

- Istnieje tylko jedna wersja destruktora na klasę
- Jeżeli nie będzie jawnie zdefiniowana, kompilator wygeneruje trywialną jego wersję
- Nigdy nie jest jawnie wywoływany

LISTA INICJALIZACYJNA KONSTRUKTORA

- (*member initializer lists*)
- Jeden ze sposobów inicjalizacji składowych klasy
- Każdy konstruktor może inicjować składowe w sposób:

```
class ArrayInt {  
    int* m_array;  
    int m_size;  
  
public:  
    ArratInt(int n) : m_size(n)    //składowa m_size jest inicjowana wartoscia m  
    {  
        m_array = new int[n];  
    }  
  
    ArratInt()  
        : m_array(nullptr), //wskaźnik m_array jest ustawiany na wartosc nullptr,  
          m_size(n)          // składowa m_size jest inicjowana wartoscia m  
    {}  
    ...  
};
```


INICJALIZACJA SKŁADOWYCH KLASY

- Domyślna inicjalizacja składowych nie-statycznych (*default member initialization*)

```
class Type
{
    float x = 0.0f;           // inicjalizacja prostej zmiennej
    int arr[3] = {1, 2, 3};    // inicjalizacja tablicy
    const int a = 42;          // inicjalizacja stalej
    Node n{...};               // inicjalizacja zmiennej złożonej (klasy)

public:
    Type();
    ...
};
```

- Nie jest to „lubiany” sposób inicjalizacji zmiennych składowych!

SKŁADOWE STAŁE I REFERENCYJNE

```
class ArrayInt {  
    int* m_array;  
    int m_size;  
    const int m_max_capacity = 255;  
    int& ref_var = m_size;  
  
public:  
    ArratInt(int n) : m_max_capacity(255), ref_var(m_size)  
    {  
        m_array = new int[n]; m_size = n;  
    }  
  
    ~ArrayInt() : m_max_capacity(255), ref_var(m_size)  
    {  
        delete[] m_array;  
    }  
};
```

The diagram illustrates the initialization of static and reference variables in the `ArrayInt` class. It shows the class definition with a static member `m_max_capacity` and a reference member `ref_var`. Both are initialized in the constructor and destructor's initialization lists. Arrows point from these initialization lists to a text box on the right, indicating that these variables must be initialized.

Składowa stała (oraz referencyjna) musi być zainicjalizowana za pomocą listy inicjalizacyjnej konstruktora lub przez wartość domyślną

METODY NIEMODYFIKUJACE (**const**)

- Metody, które gwarantują, że nie zmienią wartości żadnej składowej obiektu

```
class ArrayInt {  
    int* m_array;  
    int m_size;  
  
public:  
    ArrayInt(int n) {  
        m_array = new int[n]; m_size = n;  
    }  
  
    ~ArrayInt() {  
        delete[] m_array;  
    }  
  
    int m_size() const { return m_size; }  
};
```

- Metoda `const` nie może przypisać nowej wartości żadnej składowej klasy (ani bezpośrednio ani pośrednio)

SKŁADOWE STATYCZNE

- (*static member*)
- Składowe statyczne nie są przypisane do konkretnego obiektu
- Są one współdzielone przez wszystkie obiekty
- Mogą to być metody i pola
- Mogą być wywoływane na rzecz samej klasy (bez obecności obiektu)
- Mają czas życia programu

```
class Typ {  
public:  
    static int n;  
    static int k;  
    static void func();  
    ...  
};
```

```
int Typ::k; // składowa statyczna musi być zdefiniowana poza klasą (brak słowa static)  
int Typ::n = 3; // może być równocześnie zainicjalizowana
```

```
int main()  
{  
    cout << Typ::n << endl;  
    Typ t;  
    cout << t.n << endl;  
}
```