

# Podstawy Informatyki

**Katedra Telekomunikacji, EiT**

dr inż. Jarosław Bułat (c)

[kwant@agh.edu.pl](mailto:kwant@agh.edu.pl)

# Plan prezentacji

- » Przeciążenie/przeładowanie
- » Referencja
- » Konstruktor kopiujący
- » Dziedziczenie
- » Konstruktor klasy bazowej
- » wskaźnik *this*

# Przeciążenie czy przeładowanie?

ang. overloading

# Przeciążenie

- » Możliwość zdefiniowania wielu funkcji/metod o tych samych nazwach w ramach tego samego zasięgu:
  - wspólna nazwa
  - różne argumenty
  - **różne implementacje**
  - wykonywana implementacja zależna jest od sposobu wywołania
  - przykład statycznego **polimorfizmu**

**polimorfizm** (*wielopostaciowość*) Pozwala programiście używać zmiennych, funkcji, wartości na kilka różnych sposobów:

**jeden interfejs dla wielu różnych rzeczy**

# Przeciążenie

- » Dwie **metody**, takie same nazwy, różne argumenty (ten sam zasięg)

```
#include <stdio.h>
```

```
class MyPrint {  
    public:  
        void print(int var) {  
            printf("int: %d\n", var);  
        }  
  
        void print(float var) {  
            printf("float: %f\n", var);  
        }  
};
```

```
int main() {  
    MyPrint p;  
    p.print(123);  
    p.print(0.12f);  
    p.print(0.12);  
}
```

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
    public:  
        void print(int var) {  
            printf("int: %d\n", var);  
        }  
  
        void print(float var) {  
            printf("float: %f\n", var);  
        }  
};
```

```
int main() {  
    MyPrint p;  
    p.print(123);  
    p.print(0.12f);  
    p.print(0.12);  
}
```

- » Dwie metody, takie same nazwy, różne argumenty (ten sam zasięg)
- » Implementacja obu metod różni się

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
public:  
    void print(int var) {  
        printf("int: %d\n", var);  
    }  
  
    void print(float var) {  
        printf("float: %f\n", var);  
    }  
};
```

```
int main() {  
    MyPrint p;  
    p.print(123);  
    p.print(0.12f);  
    p.print(0.12);  
}
```

- » Dwie metody, takie same nazwy, różne argumenty (ten sam zasięg)
- » Implementacja obu metod różni się
- » Podczas wywołania wybierana jest implementacja pasująca do wzorca argumentów

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
public:  
    void print(int var) {  
        printf("int: %d\n", var);  
    }  
  
    void print(float var) {  
        printf("float: %f\n", var);  
    }  
};
```

```
int main() {  
    MyPrint p;  
    p.print(123);  
    p.print(0.12f);  
    p.print(0.12);  
}
```

- » Dwie metody, takie same nazwy, różne argumenty (ten sam zasięg)
- » Implementacja obu metod różni się
- » Podczas wywołania wybierana jest implementacja pasująca do wzorca argumentów



# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
public:  
    void print(int var) {  
        printf("int: %d\n", var);  
    }  
  
    void print(float var) {  
        printf("float: %f\n", var);  
    }  
};
```

```
int main() {  
    MyPrint p;  
    p.print(123);  
    p.print(0.12f);  
    p.print(0.12);  
}
```

- » Dwie metody, takie same nazwy, różne argumenty (ten sam zasięg)
- » Implementacja obu metod różni się
- » Podczas wywołania wybierana jest implementacja pasująca do wzorca argumentów

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
public:  
    void print(int var) {  
        printf("int: %d\n", var);  
    }  
  
    void print(float var) {  
        printf("float: %f\n", var);  
    }  
};
```

```
int main() {  
    MyPrint p;  
    p.print(123);  
    p.print(0.12f);  
    p.print(0.12);  
}
```

- » Dwie **metody**, takie same nazwy, różne argumenty (ten sam zasięg)
- » Implementacja obu metod różni się
- » Podczas wywołania wybierana jest implementacja pasująca do wzorca argumentów

ex1.cc:17:17: error: call of overloaded  
'print(double)' is ambiguous

p.print(0.12);

ex1.cc:5:14: note: **candidate:** void  
void print(int var){

ex1.cc:9:14: note: **candidate:** void  
void print(float var){

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
    public:  
        void print(void);  
        void print(int a);  
        void print(float a);  
        void print(int a, int b);  
        void print(char a, int b);  
        void print(int a, char b);  
};
```

» W języku C++ metodę lub funkcję można przeciążyć po:

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
    public:  
        void print(void);  
        void print(int a);  
        void print(float a);  
        void print(int a, int b);  
        void print(char a, int b);  
        void print(int a, char b);  
};
```

- » W języku C++ metodę lub funkcję można przeciążyć po:
  - typie argumentów

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
    public:  
        void print(void);  
        void print(int a);  
        void print(float a);  
        void print(int a, int b);  
        void print(char a, int b);  
        void print(int a, char b);  
};
```

- » W języku C++ metodę lub funkcję można przeciążyć po:
- typie argumentów
  - liczbie argumentów

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
    public:  
        void print(void);  
        void print(int a);  
        void print(float a);  
        void print(int a, int b);  
        void print(char a, int b);  
        void print(int a, char b);  
};
```

- » W języku C++ metodę lub funkcję można przeciążyć po:
- typie argumentów
  - liczbie argumentów
  - kolejności argumentów

# Przeciążenie

```
#include <stdio.h>
```

```
class MyPrint {  
    public:  
        void print(void);  
        void print(int a);  
        void print(float a);  
        void print(int a, int b);  
        void print(char a, int b);  
        void print(int a, char b);  
};
```

- » W języku C++ metodę lub funkcję można przeciążyć po:
  - typie argumentów
  - liczbie argumentów
  - kolejności argumentów
- » Nie można przeciążyć po:
  - nazwie argumentu
  - zwracanym typie
  - zwracanej liczbie wartości

# Przeciążenie konstruktora

» Uniwersalna klasa **Vector**

```
#include <stdio.h>
```

```
class Vector {  
    public:  
        Vector(double);  
        Vector(double, double);  
        Vector(double, double, double);  
        double abs(void);  
  
    private:  
        double *data_;  
        size_t dimension_;  
};
```



# Przeciążenie konstruktora

```
#include <stdio.h>
```

```
class Vector {  
    public:  
        Vector(double);  
        Vector(double, double);  
        Vector(double, double, double);  
        double abs(void);  
  
    private:  
        double *data_;  
        size_t dimension_;  
};
```

- » Uniwersalna klasa **Vector**
- » Obiekt tej klasy mogą powołać na **trzy sposoby**

# Przeciążenie konstruktora

```
#include <stdio.h>
```

```
class Vector {  
    public:  
        Vector(double);  
        Vector(double, double);  
        Vector(double, double, double);  
        double abs(void);  
  
    private:  
        double *data_;  
        size_t dimension_;  
};  
  
int main() {  
    Vector r1(0.0);           // R1  
    Vector r2(1.0, 1.0);      // R2  
    Vector r3(0.1, 0.2, 0.3); // R3  
}
```

- » Uniwersalna klasa **Vector**
- » Obiekt tej klasy mogą powołać na **trzy sposoby** w zależności od użytego konstruktora w przestrzeni odpowiednio:
  - $R^1$
  - $R^2$
  - $R^3$

# Przeciążenie konstruktora

```
#include <stdio.h>
```

```
class Vector {  
public:  
    Vector(double);  
    Vector(double, double);  
    Vector(double, double, double);  
    double abs(void);
```

```
private:  
    double *data_;  
    size_t dimension_;
```

```
};
```

```
int main() {  
    Vector r1(0.0);           // R1  
    Vector r2(1.0, 1.0);      // R2  
    Vector r3(0.1, 0.2, 0.3); // R3  
}
```

- » Uniwersalna klasa **Vector**
- » Obiekt tej klasy mogą powołać na **trzy sposoby** w zależności od użytego konstruktora w przestrzeni odpowiednio:
  - $R^1$
  - $R^2$
  - $R^3$
- » Każdy konstruktor zarezerwuje miejsce na dane w **data\_** oraz zachowa rozmiar przestrzeni
- » **abs(...)** zwróci poprawną wartość w zależności od danych i wymiaru przestrzeni

# Przeciążenie konstruktora

```
#include <stdio.h>
```

```
class Vector {  
    public:  
        Vector(double);  
        Vector(double, double);  
        Vector(double, double, double);  
        double abs(void);  
  
    private:  
        double *data_;  
        size_t dimension_;  
};
```

```
int main() {  
    Vector r1(0.0);           // R1  
    Vector r2(1.0, 1.0);      // R2  
    Vector r3(0.1, 0.2, 0.3); // R3  
}
```

- » Przykład polimorfizmu
- » Jedna klasa
- » Przeciążony konstruktor pozwala utworzyć obiekty w różnych przestrzeniach
- » Jedna metoda oblicza wynik na różne sposoby, w zależności od danych (w tym przypadku od rozmiaru przestrzeni)



quiz

**PI10\_overl**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

# Referencja vs wskaźnik

czym się różnią?

# Referencja

```
#include <iostream>
using namespace std;

int main() {
    int a = 0;
    int &b = a;

    cout << a << endl;
    b = 10;
    cout << a << endl;
    cout << b << endl;
}
```

- » Zmienną referencyjną można traktować jak drugą nazwę dla zmiennej (lub jej alias)
- » Działanie podobne do zmiennej wskaźnikowej ale z wieloma ograniczeniami - jest bezpieczniejsza w użyciu
- » Skoro **b** i **a** są tym samym to wynik działania:  
0  
10  
10
- » Łatwo używać (brak \*&)

# Referencja

```
#include <iostream>
using namespace std;
```

```
int main() {
    int a = 0;
    int &b = a;

    cout << a << endl;
    b = 10;
    cout << a << endl;
    cout << b << endl;

    int &c;
}
```

- » Zmienna referencyjna musi być zainicjalizowana
- error: 'c' declared as reference but not initialized
- int &c;



# Referencja

```
#include <iostream>
using namespace std;
```

```
int main() {
    int a = 0;
    int &b = a;

    cout << a << endl;
    b = 10;
    cout << a << endl;
    cout << b << endl;

    int c;
    &b = c;
}
```

- » Zmienną referencyjną nie można ponownie zainicjalizować (przypisać do innej zmiennej)
- nie można jej więc “przestawić” na nieistniejącą zmienną (jak wskaźnik)
  - nie można użyć na nieistniejącym obiekcie (nie może być NULL)

# Referencja - przykład

```
#include <iostream>
using namespace std;

void ref(int &arg) {
    arg++;
}

int main() {
    int a = 0;

    cout << a << endl;
    ref(a);
    cout << a << endl;
}
```

- » Najczęściej wykorzystuje się dla argumentów funkcji
- » Zmienna **arg** w funkcji jest tym samym czym zmienna **a**
- » Zmiany zmiennej **arg** są odzwierciedlane na zmiennej **a**
- » Nie ma kopiowania przez argument (szybko przy dużych obiektach)
- » Rezultat: 0\n 1\n
- » **Zaleta:** łatwe w użyciu + bezp.
- » **Wada:** patrząc na wywołanie nie wiem czy przez kopię czy przez referencję

# Referencja - przykład

```
#include <iostream>
using namespace std;

int ref(const int &arg) {
    return arg+1;
}

int main() {
    int a = 0;

    cout << a << endl;
    cout << ref(a) << endl;
}
```

- » Można **zabronić modyfikacji** “referowanej” zmiennej
- » Rezultat będzie taki sam jak podczas przekazywania argumentu przez kopię
- » Zaleta użycia referencji to brak kopiowania (szybkie przy dużych obiektach)

# Referencja - porównanie

```
#include <iostream>
using namespace std;

int ref(const int &arg) {
    return arg+1;
}

int main() {
    int a = 0;

    cout << a << endl;
    cout << ref(a) << endl;
    cout << a << endl;
}
```

```
#include <iostream>
using namespace std;

void ref(int *arg) {
    (*arg)++;
    // *arg++; ERROR !!!
}

int main() {
    int a = 0;

    cout << a << endl;
    ref(&a);
    cout << a << endl;
}
```

- » Wykorzystanie referencji jest prostsze niż wskaźników
- » Mniejsza możliwość popełnienia błędu



quiz

**PI10\_ref**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

# Problem

```
class Vector {  
    public:  
        double *data_;  
        Vector(double data) {  
            data_ = new double;  
            *data_ = data;  
        }  
        ~Vector() {delete data_;}  
        double add(Vector arg){  
            return *data_+*(arg.data_);  
        }  
};
```

```
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

- » Intencją jest utworzenie klasy, **dodającej** N-wymiarowe wektory (na razie  $N=1$ )
- » Klasa dynamicznie rezerwuje i zwalnia pamięć dla danych
- » Metoda **add(...)** wykonuje dodawanie dwóch zmiennych

# Problem

```
class Vector {  
    public:  
        double *data_;  
        Vector(double data) {  
            data_ = new double;  
            *data_ = data;  
        }  
        ~Vector() {delete data_;}  
        double add(Vector arg){  
            return *data_ + *(arg.data_);  
        }  
};  
  
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

- » Intencją jest utworzenie klasy, **dodającej** N-wymiarowe wektory (na razie  $N=1$ )
- » Klasa dynamicznie rezerwuje i zwalnia pamięć dla danych
- » Metoda **add(...)** wykonuje dodawanie dwóch zmiennych
  - **pierwszy składnik** to dane z obiektu tej klasy
  - **drugi składnik** to dane z obiektu przekazanego przez argument

# Problem

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
    }
    ~Vector() {delete data_;}
    double add(Vector arg){
        return *data_+*(arg.data_);
    }
};

int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
```

» Problem:

1

\*\*\* Error in

`/PI/lab\_10\_overload\_inheritance/ex  
04': **double free or corruption**

(fasttop): 0x0000000002245c40 \*\*\*

===== Backtrace: =====



# Problem

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
    }  
    ~Vector() {delete data_;}  
    double add(Vector arg){  
        return *data_+*(arg.data_);  
    }  
};  
  
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

» Problem:

1

\*\*\* Error in  
`/PI/lab\_10\_overload\_inheritance/ex  
04': **double free or corruption**  
(fasttop): 0x0000000002245c40 \*\*\*  
===== Backtrace: =====

- » Program poprawnie zadziałał ale  
potem wykonał niedozwoloną  
operację
- » Dlaczego?

# Problem

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
        cout << data_ << endl;  
    }  
    ~Vector() {  
        cout << data_ << endl;  
        delete data_;  
    }  
    double add(Vector arg){  
        return *data_ + *(arg.data_);  
    }  
};  
  
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

1. Konstruktor a: New: 0x861c20

# Problem

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
        cout << data_ << endl;  
    }  
    ~Vector() {  
        cout << data_ << endl;  
        delete data_;  
    }  
    double add(Vector arg){  
        return *data_+*(arg.data_);  
    }  
};
```

```
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

1. Konstruktor a: **New: 0x861c20**

2. Konstruktor b: **New: 0x862c50**

# Problem

```

class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
        cout << data_ << endl;
    }
    ~Vector() {
        cout << data_ << endl;
        delete data_;
    }
    double add(Vector arg){
        return *data_ + *(arg.data_);
    }
};

int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
  
```

1. Konstruktor **a**: **New: 0x861c20**
2. Konstruktor **b**: **New: 0x862c50**
3. Metoda **add(...)** otrzymuje **kopię zmiennej (obektu) b**

niejawnie wykonywana zostanie operacja przypisania:

**-Vector arg(b);**

w przypadku obiektów, skopiowane zostaną składowe b.\* do odpowiednich składowych arg.\*  
czyli: **arg.data\_ = b.data\_**

**tylko wskaźniki zostaną skopiowane**

# Problem

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
        cout << data_ << endl;
    }
    ~Vector() {
        cout << data_ << endl;
        delete data_;
    }
    double add(Vector arg){
        return *data_+*(arg.data_);
    }
};

int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
```

1. Konstruktor **a**: **New: 0x861c20**
2. Konstruktor **b**: **New: 0x862c50**
3. Metoda **add(...)** otrzymuje **kopię zmiennej (obiektu) b**
  - a. **arg** jest kopią **b**, wartość **data\_**: **0x862c50** (czyli taka jak w obiekcie **b**)

# Problem

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
        cout << data_ << endl;
    }
    ~Vector() {
        cout << data_ << endl;
        delete data_;
    }
    double add(Vector arg){
        return *data_+*(arg.data_);
    }
};

int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
```

1. Konstruktor **a**: **New: 0x861c20**
2. Konstruktor **b**: **New: 0x862c50**
3. Metoda **add(...)** otrzymuje **kopię zmiennej (obektu) b**
  - a. **arg** jest kopią **b**, wartość **data\_**: **0x862c50** (czyli taka jak w obiekcie **b**)

# Problem

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
        cout << data_ << endl;  
    }  
    ~Vector() {  
        cout << data_ << endl;  
        delete data_;  
    }  
    double add(Vector arg){  
        return *data_+*(arg.data_);  
    }  
};  
  
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

1. Konstruktor **a**: **New: 0x861c20**
2. Konstruktor **b**: **New: 0x862c50**
3. Metoda **add(...)** otrzymuje **kopię zmiennej (obektu) b**
  - a. **arg** jest kopią **b**, wartość **data\_**: **0x862c50** (czyli taka jak w obiekcie **b**)
  - b. wykonuje dodawanie
  - c. niszczy obiekt **arg** (wychodzi z zasięgu)

# Problem

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
        cout << data_ << endl;
    }
    ~Vector() {
        cout << data_ << endl;
        delete data_;
    }
    double add(Vector arg){
        return *data_+*(arg.data_);
    }
};

int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
```

1. Konstruktor **a**: **New: 0x861c20**
2. Konstruktor **b**: **New: 0x862c50**
3. Metoda **add(...)** otrzymuje **kopię zmiennej (obektu) b**
  - a. **arg** jest kopią **b**, wartość **data\_**: **0x862c50** (czyli taka jak w obiekcie **b**)
  - b. wykonuje dodawanie
  - c. niszczy obiekt **arg** (wychodzi z zasięgu)
  - d. **arg** jest obiektem więc wykonywany jest destruktork



# Problem

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
        cout << data_ << endl;
    }
    ~Vector() {
        cout << data_ << endl;
        delete data_;
    }
    double add(Vector arg){
        return *data_+*(arg.data_);
    }
};

int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
```

1. Konstruktor **a**: **New: 0x861c20**
2. Konstruktor **b**: **New: 0x862c50**
3. Metoda **add(...)** otrzymuje **kopię zmiennej (obektu) b**
  - a. **arg** jest kopią **b**, wartość **data\_**: **0x862c50** (czyli taka jak w obiekcie **b**)
  - b. wykonuje dodawanie
  - c. niszczy obiekt **arg** (wychodzi z zasięgu)
  - d. **arg** jest obiektem więc wykonywany jest destruktory: **zwalnianie 0x862c50**

# Problem

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
        cout << data_ << endl;
    }
    ~Vector() {
        cout << data_ << endl;
        delete data_;
    }
    double add(Vector arg){
        return *data_+*(arg.data_);
    }
};

int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
```

1. Konstruktor **a**: **New: 0x861c20**
2. Konstruktor **b**: **New: 0x862c50**
3. Metoda **add(...)** otrzymuje **kopię zmiennej (obektu) b**
  - a. **arg** jest kopią **b**, wartość **data\_**: **0x862c50** (czyli taka jak w obiekcie **b**)
  - b. wykonuje dodawanie
  - c. niszczy obiekt **arg** (wychodzi z zasięgu)
  - d. **arg** jest obiektem więc wykonywany jest destruktory: zwalnia **0x862c50**
4. Niszczony jest obiekt **b** czyli **zwalniana pamięć 0x862c50 !!!!!**

# Problem

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
        cout << data_ << endl;
    }
    ~Vector() {
        cout << data_ << endl;
        delete data_;
    }
    double add(Vector arg){
        return *data_+*(arg.data_);
    }
};

int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
```

1. Konstruktor **a**: **New: 0x861c20**
2. Konstruktor **b**: **New: 0x862c50**
3. **Rezultat:**
  - a. dwukrotnie zwalniana pamięć o adresie **0x862c50**
  - b. podczas drugiego wywołania OS wykrywa niedozwoloną operację i zgłasza wyjątek
  - c. nie dochodzi do wykonania destruktora na obiekcie **a** ponieważ proces został zatrzymany

# Rozwiązanie

» Przekaż **wskaźnik** zamiast obiektu

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
        cout << data_ << endl;  
    }  
    ~Vector() {  
        cout << data_ << endl;  
        delete data_;  
    }  
    double add(Vector arg){  
        return *data_+*(arg.data_);  
    }  
};
```

```
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

# Rozwiązanie

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
        cout << data_ << endl;  
    }  
    ~Vector() {  
        cout << data_ << endl;  
        delete data_;  
    }  
    double add(Vector arg){  
        return *data_+*(arg.data_);  
    }  
};
```

```
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

- » Przekaż wskaźnik zamiast obiektu
- » Przekaż **referencję** zamiast obiektu

# Rozwiązanie

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
        cout << data_ << endl;  
    }  
    ~Vector() {  
        cout << data_ << endl;  
        delete data_;  
    }  
    double add(Vector arg){  
        return *data_+*(arg.data_);  
    }  
};
```

```
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

- » Przekaż wskaźnik zamiast obiektu
- » Przekaż referencję zamiast obiektu
- » Utwórz konstruktor kopiujący

# Rozwiązanie - wskaźnik

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
    }
    ~Vector() {
        delete data_;
    }
    double add(Vector arg){
        return *data_+*(arg.data_);
    }
};
```

```
int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(b) << endl;
}
```

```
class Vector {
public:
    double *data_;
    Vector(double data) {
        data_ = new double;
        *data_ = data;
    }
    ~Vector() {
        delete data_;
    }
    double add(Vector *arg){
        return *data_+*(arg->data_);
    }
};
```

```
int main() {
    Vector a(0.0);
    Vector b(1.0);
    cout << a.add(&b) << endl;
}
```

# Rozwiązanie - referencja

- » Minimalne różnice w stosunku do oryginalnego kodu
- » Duża czytelność kodu
- » Referencja (jak i wskaźnik) jest niszczone na końcu bloku (końcu metody add()) ale to **nie wyzwala destruktora oryginalnego obiektu**

```
class Vector {  
    public:  
        double *data_;  
        Vector(double data) {  
            data_ = new double;  
            *data_ = data;  
        }  
        ~Vector() {  
            delete data_;  
        }  
        double add(Vector &arg) {  
            return *data_ + *(arg.data_);  
        }  
};
```

```
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```



# Rozwiązanie - konstruktor kopiujący

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
    }  
    ~Vector() {delete data_;}  
    double add(Vector arg){  
        return *data_+*(arg.data_);  
    }  
};
```

```
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

» Dodaj **konstruktor kopiujący** aby podczas wyrażenia

**Vector** arg(b);

poprawnie utworzył nowy obiekt

# Rozwiązanie - konstruktor kopiujący

```
class Vector {  
public:  
    double *data_;  
    Vector(double data) {  
        data_ = new double;  
        *data_ = data;  
    }  
    Vector(const Vector &copy);  
    ~Vector() {delete data_;}  
    double add(Vector arg){  
        return *data_+*(arg.data_);  
    }  
};
```

```
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

## » Konstruktor kopiujący:

- zarezerwuj nową pamięć
- skopiuj zawartość danych z kopiowanego obiektu do nowego

```
Vector::Vector(const Vector &copy){  
    data_ = new double;  
    memcpy(data_, copy.data_, sizeof(double));  
}
```

- » W rezultacie, podczas tworzenia obiektu **arg** zostanie przydzielona nowa pamięć a zawartość danych z obiektu **b** zostanie do niego skopiowana

# Rozwiązanie - konstruktor kopiujący

```
class Vector {  
    public:  
        double *data_;  
        Vector(double data) {  
            data_ = new double;  
            *data_ = data;  
        }  
        Vector(const Vector &copy);  
        ~Vector() {delete data_;}  
        double add(Vector arg){  
            return *data_+*(arg.data_);  
        }  
};  
  
int main() {  
    Vector a(0.0);  
    Vector b(1.0);  
    cout << a.add(b) << endl;  
}
```

- » Konstruktor kopiujący to specjalny konstruktor
- » Zawsze przyjmuje tylko jeden argument - referencję do typu klasy w której jest zdefiniowany
- » Jeżeli sami go nie utworzymy, zrobi to automatycznie kompilator ale ograniczy się do kopiowania składowych
- » Konstruktor jest wywoływany jeżeli inicjalizujemy obiekt innym obiektem tej samej klasy:  
T obj0;  
T obj1(obj0);



quiz

**PI10\_cc**

**socrative.com**

- login
- student login

Room name:

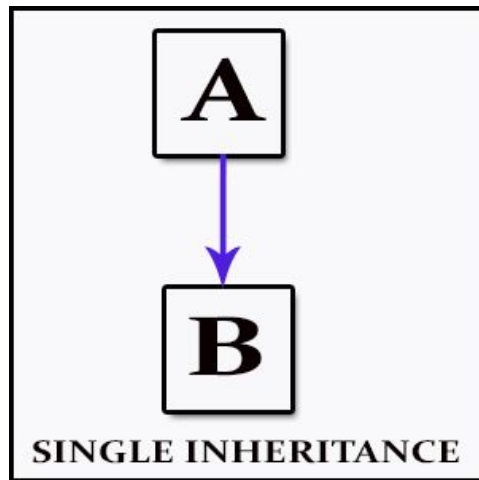
**KWANTAGH**

# Dziedziczenie

esencja OOP

# Dziedziczenie

- » Mechanizm współdzielenia funkcjonalności pomiędzy klasami czyli ponowne użycie kodu
- » Jeżeli dwie klasy wykonują podobne czynności, możemy **wyekstrahować wspólną część** i uczynić ją klasą bazową po której będą dziedziczyć obie klasy
- » **A** jest klasą podstawową, po której dziedziczy klasa **B** (klasa pochodna)
- » Klasa **B** **rozszerza** klasę **A**
- » W obiektach klasy **B** możemy wykorzystywać składowe (zmienne, metody) klasy **A**
- » Z jednej klasy bazowej możemy uzyskać wiele klas pochodnych
- » C++ pozwala na wielokrotne dziedziczenie: po kilku klasach podstawowych

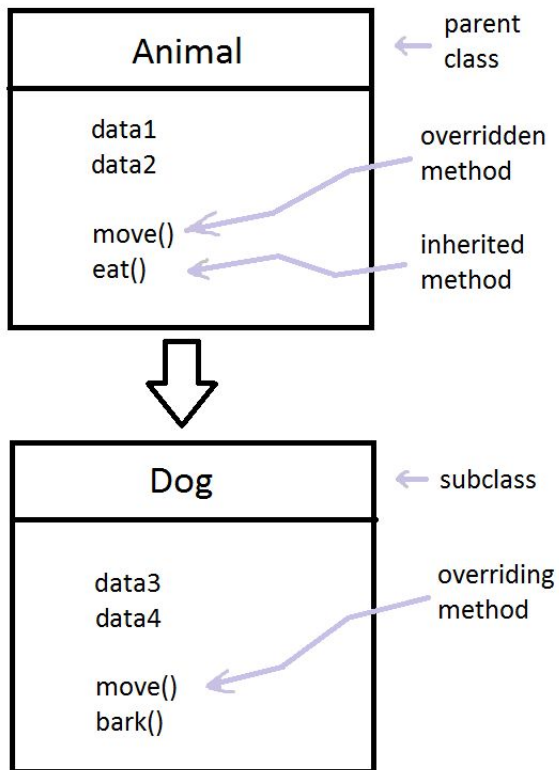


# Dziedziczenie

- » Zależność pomiędzy klasami bazowymi i pochodnymi tworzy hierarchię klas
- » Hierarchia klas jest niezmienna - jest ustalana podczas kompilacji
- » Klasy pochodne otrzymają atrybuty klasy podstawowej, mogą dodawać własne składowe
- » Dopuszczalne jest nadpisywanie metod (zmianę implementacji) klasy podstawowej
- » Istnieje możliwość ograniczenie widoczności składowych klasy podstawowej w klasie pochodnej

# Dziedziczenie

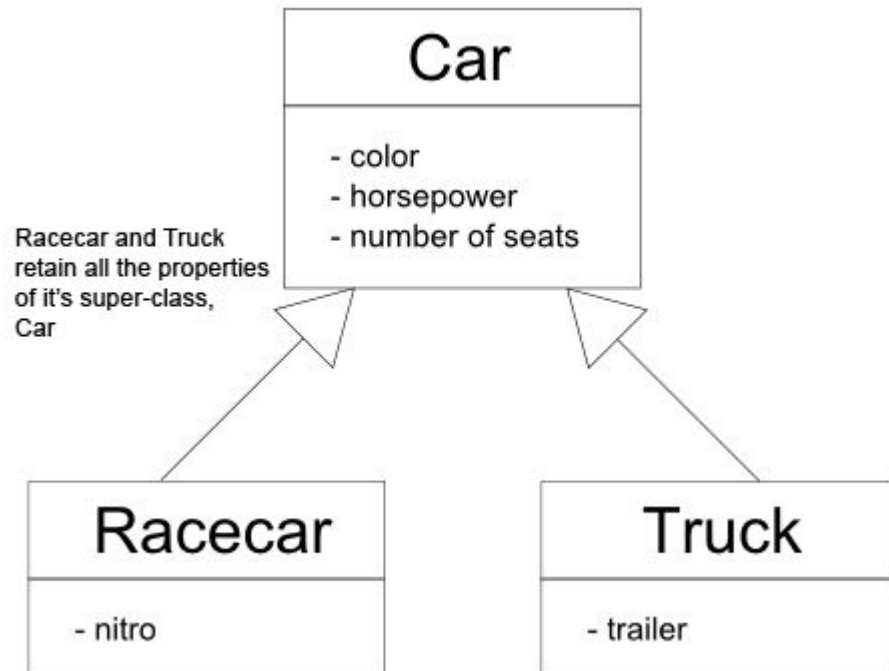
- » Animal: klasa podstawowa (bazowa)
- » Dog: klasa pochodna





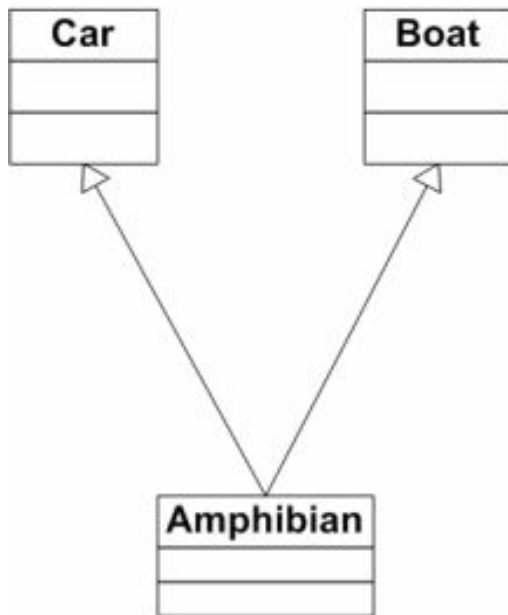
# Dziedziczenie

- » Dwie klasy pochodne z jednej klasy bazowej



# Dziedziczenie

## Multiple Inheritance



- » Dziedziczenie wielokrotne
- » Dwie klasy podstawowe
- » Klasa pochodna może używać funkcjonalności obu



quiz

**PI10\_001**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

# Dziedziczenie

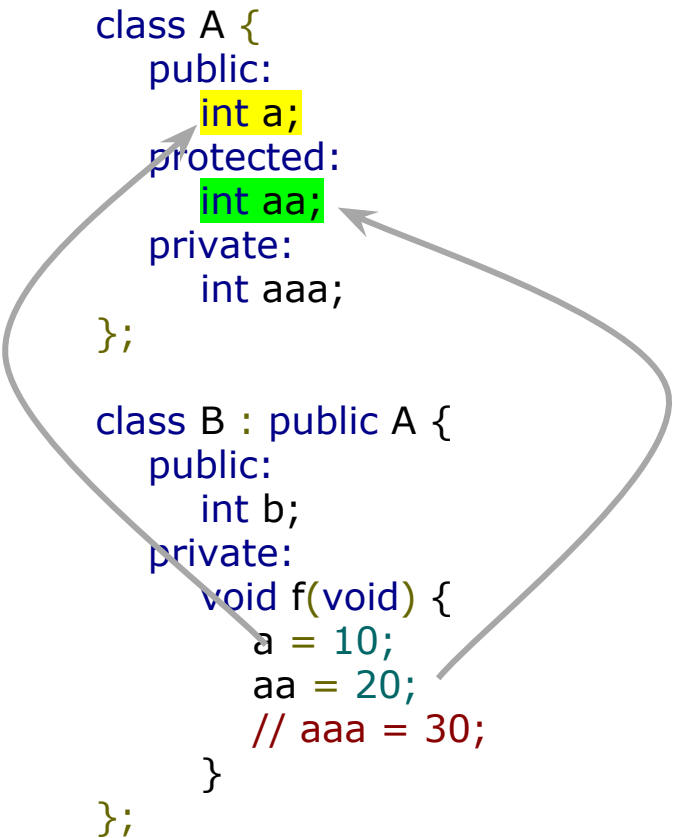
```
class A {  
    public:  
        int a;  
    protected:  
        int aa;  
    private:  
        int aaa;  
};
```

```
class B : public A {  
    public:  
        int b;  
    private:  
        void f(void) {  
            a = 10;  
            aa = 20;  
            // aaa = 30;  
        }  
};
```

- » Klasa **B** (pochodna) dziedziczy po klasie **A** (podstawowej)
- » Klasa podstawowa musi być wcześniej zadeklarowana

# Dziedziczenie

```
class A {  
    public:  
        int a;  
    protected:  
        int aa;  
    private:  
        int aaa;  
};  
  
class B : public A {  
    public:  
        int b;  
    private:  
        void f(void) {  
            a = 10;  
            aa = 20;  
            // aaa = 30;  
        }  
};
```

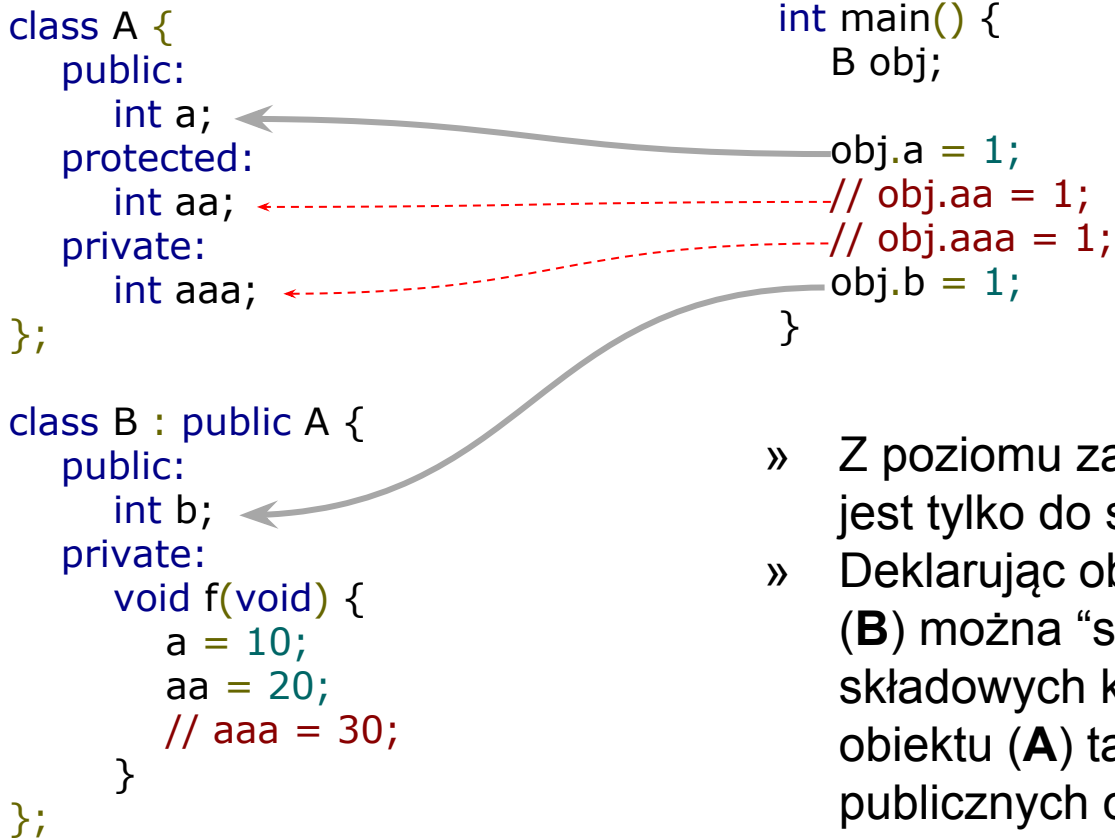


- » Klasa **B** (pochodna) dziedziczy po klasie **A** (podstawowej)
- » Klasa podstawowa musi być wcześniej zdefiniowana
- » Dowolne metody klasy pochodnej mogą odnosić się do dowolnych składowych **publicznych** i **chronionych** klasy bazowej, tak jak do swoich składowych
- » Dostępu do składowych prywatnych nie ma

# Dziedziczenie

```
class A {  
    public:  
        int a;  
    protected:  
        int aa;  
    private:  
        int aaa;  
};  
  
class B : public A {  
    public:  
        int b;  
    private:  
        void f(void) {  
            a = 10;  
            aa = 20;  
            // aaa = 30;  
        }  
};
```

**int main() {**  
 B obj;  
  
 obj.a = 1;  
 // obj.aa = 1;  
 // obj.aaa = 1;  
 obj.b = 1;  
}



- » Z poziomu zasięgu obiektu, dostęp jest tylko do składowych publicznych
- » Deklarując obiekt klasy pochodnej (**B**) można “sięgnąć” publicznych składowych klasy podstawowej tego obiektu (**A**) tak jak składowych publicznych obiektu **B**

# Dziedziczenie - przykład

```
#include <iostream>
using namespace std;
```

```
class CDRM {
public:
    void eject();
    char *read();
};

class CDRW : public CDRM {
public:
    void write(char *);
};
```

```
int main() {
    char tab[650];
    CDRM cd;
    cd.eject();
    cd.read();

    CDRW cdrw;
    cdrw.eject();
    cdrw.read();
    cdrw.write(tab);
}
```

» Klasa **pochodna** używa implementacji z klasy **podstawowej**

# Dziedziczenie - przykład

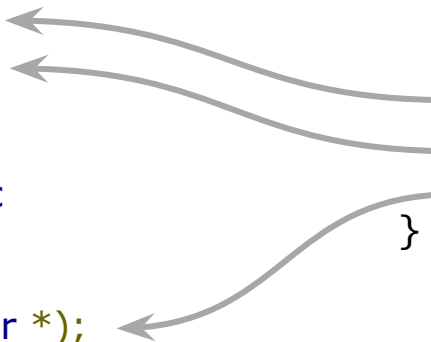
```
#include <iostream>
using namespace std;
```

```
class CDROM {
public:
    void eject();
    char *read();
};
```

```
class CDRW : public
CDROM{
public:
    void write(char *);
};
```

```
int main() {
    char tab[650];
    CDROM cd;
    cd.eject();
    cd.read();

    CDRW cdrw;
    cdrw.eject();
    cdrw.read();
    cdrw.write(tab);
}
```



» Klasa pochodna używa implementacji z klasy podstawowej



# Dziedziczenie - przykład

```
#include <iostream>
using namespace std;
```

```
class CDROM {
public:
    void eject();
    char *read();
};
```

```
class CDRW : public
CDROM{
public:
    void write(char *);
    char *read();
};
```

```
int main() {
    char tab[650];
    CDROM cd;
    cd.eject();
    cd.read();

    CDRW cdrw;
    cdrw.eject();
    cdrw.read();
    cdrw.write(tab);
}
```

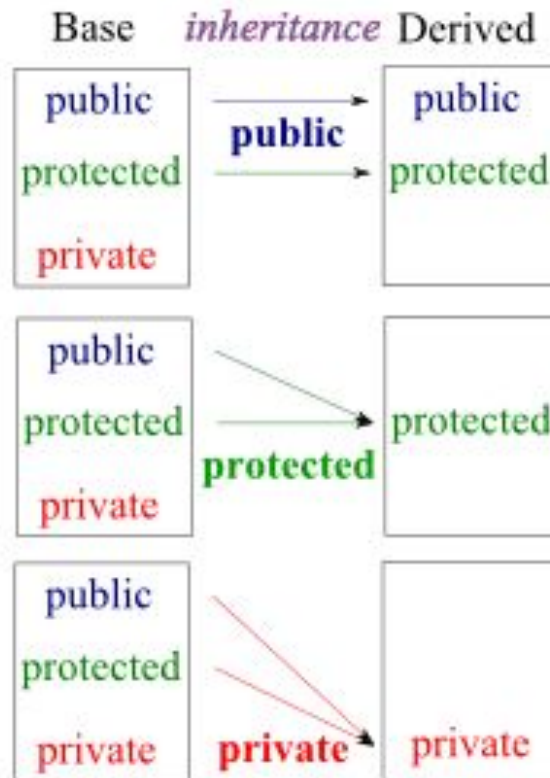
- » Klasa pochodna może “przykryć” definicje komponentu klasy bazowej nawet tego samego typu (bez przeciążenia)

# Dziedziczenie - przykład

```
#include <iostream>
using namespace std;

class CDROM {
public:
    void eject();
    char *read();
};

class CDRW : public
CDROM{
public:
    void write(char *);
    char *read();
};
```





quiz

**PI10\_002**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

Konstruktor mojego konstruktora  
jest moim przyjacielem :-)

# Konstruktor klasy bazowej

```
class CDROM {  
    public:  
        CDROM(){}  
        void printVendor(){  
            cout << "vendor:" << endl;  
        }  
};
```

```
class CDRW : public CDROM{  
    public:  
        CDRW(){}  
};
```

```
int main() {  
    CDRW cdrw;  
    cdrw.printVendor();  
}
```

- » Rozważmy prosty przypadek
- » Klasa bazowa
- » Klasa pochodna
- » Konstruktory "puste"

# Konstruktor klasy bazowej

```
class CDROM {  
    public:  
        CDROM(){}  
        void printVendor(){  
            cout << "vendor:" << endl;  
        }  
};
```

```
class CDRW : public CDROM{  
    public:  
        CDRW(){}  
};
```

```
int main() {  
    CDRW cdrw;  
    cdrw.printVendor();  
}
```

- » Klasa bazowa
- » Klasa pochodna
- » Konstruktory “puste”
- » Obiekt klasy tworzony bez żadnej inicjalizacji

# Konstruktor klasy bazowej

```
class CDROM {  
    public:  
        CDROM(){}  
        void printVendor(){  
            cout << "vendor:" << endl;  
        }  
};
```

```
class CDRW : public CDROM{  
    public:  
        CDRW(){}  
};
```

```
int main() {  
    CDRW cdrw;  
    cdrw.printVendor();  
}
```

- » Klasa bazowa
- » Klasa pochodna
- » Konstruktory "puste"
- » Obiekt klasy tworzony bez żadnej inicjalizacji
- » Metoda wywołana na obiekcie klasy pochodnej jest metodą klasy bazowej

# Konstruktor klasy bazowej

```
class CDROM {  
    private:  
        string vendor_;  
    public:  
        CDROM(string vendor);  
        void printVendor(){  
            cout << "vendor:" << endl;  
        }  
};
```

```
class CDRW : public CDROM{  
    public:  
        CDRW(string);  
};
```

```
int main() {  
    CDRW cdrw("TEAC");  
    cdrw.printVendor();  
}
```

- » Klasa bazowa ma **składową prywatną**
  - konstruktor ją inicjalizuje
- » Klasa pochodna nie ma dostępu do prywatnych składowych klasy bazowej



# Konstruktor klasy bazowej

```
class CDROM {  
    private:  
        string vendor_;  
    public:  
        CDROM(string vendor);  
        void printVendor(){  
            cout << "vendor:" << endl;  
        }  
};
```

```
class CDRW : public CDROM{  
    public:  
        CDRW(string);  
};
```

```
int main() {  
    CDRW cdrw("TEAC");  
    cdrw.printVendor();  
}
```

- » Klasa bazowa ma **składową prywatną**
  - konstruktor ją inicjalizuje
- » Klasa pochodna nie ma dostępu do prywatnych składowych klasy bazowej
- » Jak zainicjalizować **vendor\_** z poziomu konstruktora klasy pochodnej???

# Konstruktor klasy bazowej

```
class CDRM {  
    private:  
        string vendor_;  
    public:  
        CDRM(string vendor);  
        void printVendor(){  
            cout << "vendor:" << endl;  
        }  
};
```

CDRM::CDRM(string vendor) :  
 vendor\_(vendor){  
}

```
class CDRW : public CDRM{  
    public:  
        CDRW(string);  
};
```

CDRW::CDRW(string vendor) :  
 CDRM(vendor) {  
}

```
int main() {  
    CDRW cdrw("TEAC");  
    cdrw.printVendor();  
}
```

» Konstruktor klasy podstawowej  
inicjalizuje **vendor\_** listą  
inicjalizacyjną

# Konstruktor klasy bazowej

```
class CDROM {  
    private:  
        string vendor_;  
    public:  
        CDROM(string vendor);  
        void printVendor(){  
            cout << "vendor:" << endl;  
        }  
};
```

```
class CDRW : public CDROM{  
    public:  
        CDRW(string);  
};
```

```
int main() {  
    CDRW cdrw("TEAC");  
    cdrw.printVendor();  
}
```

```
CDROM::CDROM(string vendor) :  
    vendor_(vendor){  
}
```

```
CDRW::CDRW(string vendor) :  
    CDROM(vendor) {  
}
```

- » Konstruktor klasy podstawowej inicjalizuje `vendor_` listą inicjalizacyjną
- » **Konstruktor klasy pochodnej** wywołuje jawnie konstruktor klasy podstawowej przekazując parametr do inicjalizacji

# Konstruktor klasy bazowej

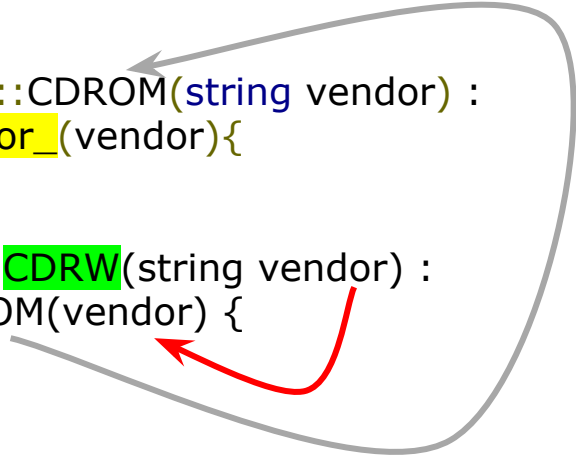
```
class CDROM {  
    private:  
        string vendor_;  
    public:  
        CDROM(string vendor);  
        void printVendor(){  
            cout << "vendor:" << endl;  
        }  
};
```

```
class CDRW : public CDROM{  
    public:  
        CDRW(string);  
};
```

```
int main() {  
    CDRW cdrw("TEAC");  
    cdrw.printVendor();  
}
```

```
CDROM::CDROM(string vendor) :  
    vendor_(vendor){  
}
```

```
CDRW::CDRW(string vendor) :  
    CDROM(vendor) {  
}
```



- » Konstruktor klasy podstawowej inicjalizuje `vendor_` listą inicjalizacyjną
- » **Konstruktor klasy pochodnej** wywołuje jawnie konstruktor klasy podstawowej przekazując parametr do inicjalizacji

# Konstruktor klasy bazowej

- » Wygląda podobnie jak lista inicjalizacyjna
- » Mogę wywołać dowolny (przeciążony) konstruktor klasy podstawowej
- » Mogę przekazać dowolną liczbę danych
- » Z poziomu klasy pochodnej mogę w pełni zainicjalizować klasę podstawową już w konstruktorze
- » Mogę jawnie wywołać konstruktor klasy podstawowej

```
CDROM::CDROM(string vendor) :  
    vendor_(vendor){  
}
```

```
CDRW::CDRW(string vendor) :  
    CDROM(vendor) {  
}
```

- » Konstruktor klasy podstawowej inicjalizuje `vendor_` listą inicjalizacyjną
- » **Konstruktor klasy pochodnej** wywołuje jawnie konstruktor klasy podstawowej przekazując parametr do inicjalizacji



quiz

**PI10\_003**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

# Wskaźnik “this”

# wskaźnik **this**

```
class CDROM {  
    public:  
        void printThis(){  
            cout << this << endl;  
        }  
};
```

```
int main() {  
    CDROM cdrom;  
    cdrom.printThis();  
    cout << &cdrom << endl;  
  
    CDROM cdrom1;  
    cdrom1.printThis();  
    cout << &cdrom1 << endl;  
}
```

- » **“this”** jest wskaźnikiem na obiekt
- » Można go użyć w metodach klasy, **this dotyczy konkretnego obiektu na którym metoda została wywołana**



# wskaźnik **this**

```
class CDROM {  
    public:  
        void printThis(){  
            cout << this << endl;  
        }  
};
```

```
int main() {  
    CDROM cdrom;  
    cdrom.printThis();  
    cout << &cdrom << endl;
```

```
    CDROM cdrom1;  
    cdrom1.printThis();  
    cout << &cdrom1 << endl;  
}
```

- » “**this**” jest wskaźnikiem na obiekt
- » Można go użyć w metodach klasy, **this** dotyczy konkretnego obiektu na którym metoda została wywołana
- » Rezultat:

0x7fff4ad39710

0x7fff4ad39710

0x7fff4ad39730

0x7fff4ad39730

# wskaźnik **this**

```
class CDROM {  
    public:  
        void printThis(){  
            cout << this << endl;  
        }  
};
```

```
int main() {  
    CDROM cdrom;  
    cdrom.printThis();  
    cout << &cdrom << endl;
```

```
    CDROM cdrom1;  
    cdrom1.printThis();  
    cout << &cdrom1 << endl;  
}
```

- » “**this**” jest wskaźnikiem na obiekt
- » Można go użyć w metodach klasy, **this** dotyczy konkretnego obiektu na którym metoda została wywołana
- » Rezultat:

```
0x7fff4ad39710  
0x7fff4ad39710  
0x7fff4ad39730  
0x7fff4ad39730
```

- » Do czego może się to przydać?

# wskaźnik **this**

- » Rozwiązywanie niejednoznacznych sytuacji

# wskaźnik **this**

```
class CDROM {  
    private:  
        string vendor; // not vendor_  
    public:  
        CDROM(string);  
};  
  
CDROM::CDROM(string vendor) {  
    this->vendor = vendor;  
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
  - w konstruktorze inicjalizuję składową

# wskaźnik **this**

```
class CDROM {  
    private:  
        string vendor; // not vendor_  
    public:  
        CDROM(string);  
};  
  
CDROM::CDROM(string vendor) {  
    this->vendor = vendor;  
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
  - w konstruktorze inicjalizuję składową
  - nazwa składowej jest taka sama jak argumentu

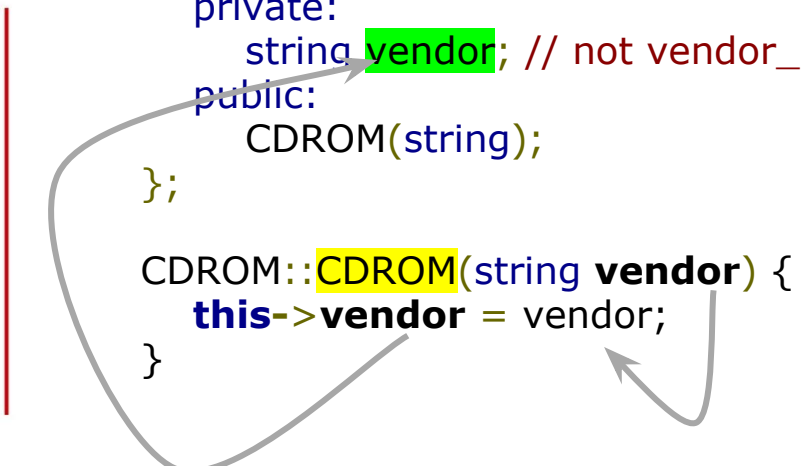
# wskaźnik **this**

```
class CDROM {  
    private:  
        string vendor; // not vendor_  
    public:  
        CDROM(string);  
};  
  
CDROM::CDROM(string vendor) {  
    this->vendor = vendor;  
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
  - w konstruktorze inicjalizuję składową
  - nazwa składowej jest taka sama jak argumentu
  - wskaźnik **this** pozwoli rozróżnić obie nazwy

# wskaźnik **this**

```
class CDROM {  
    private:  
        string vendor; // not vendor_  
    public:  
        CDROM(string);  
};  
  
CDROM::CDROM(string vendor) {  
    this->vendor = vendor;  
}
```



- » Rozwiązywanie niejednoznacznych sytuacji
- w **konstruktorze** inicjalizuję **składową**
  - nazwa składowej jest taka sama jak argumentu
  - wskaźnik **this** pozwoli rozróżnić obie nazwy
  - w tym przypadku, argument konstruktora “przykryje” nazwę składowej klasy

# wskaźnik **this**

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)



# wskaźnik **this**

```
class CDROM;

void externalFunction(CDROM
*cdromInstance){
}

class CDROM {
    public:
        CDROM();
        void callExternal(){
            externalFunction(this);
        }
};

int main() {
    CDROM cdrom;
    cdrom.callExternal();
    externalFunction(&cdrom);
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda **callExternal()** wywołuje zewnętrzną funkcję do której przekazuje wskaźnik na obiekt na którym została wywołana

# wskaźnik **this**

```
class CDROM;

void externalFunction(CDROM
*cdromInstance){
}

class CDROM {
public:
    CDROM();
    void callExternal(){
        externalFunction(this);
    }
};

int main() {
    CDROM cdrom;
    cdrom.callExternal();
    externalFunction(&cdrom);
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda callExternal() wywołuje zewnętrzną funkcję do której przekazuje wskaźnik na obiekt na którym została wywołana

# wskaźnik **this**

```
class CDROM;

void externalFunction(CDROM
*cdromInstance){
}

class CDROM {
public:
    CDROM();
    void callExternal(){
        externalFunction(this);
    }
};

int main() {
    CDROM cdrom;
    cdrom.callExternal();
    externalFunction(&cdrom);
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda callExternal() wywołuje zewnętrzną funkcję do której przekazuje wskaźnik na **obiekt** na którym została wywołana

# wskaźnik **this**

```
class CDROM;

void externalFunction(CDROM
*cdromInstance){
}

class CDROM {
public:
    CDROM();
    void callExternal(){
        externalFunction(this);
    }
};

int main() {
    CDROM cdrom;
    cdrom.callExternal();
    externalFunction(&cdrom);
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda callExternal() wywołuje zewnętrzną funkcję do której przekazuje wskaźnik na **obiekt** na którym została wywołana
- » Funkcję można też wywołać z poziomu obiektu

# wskaźnik **this**

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » **Metoda zwraca referencję do obiektu na którym została wykonana**

# Konstruktor klasy bazowej

```
class CDROM {  
    int x_;  
    int y_;  
public:  
    CDROM &setX(int x){  
        x_ = x;  
        return *this;  
    }  
    CDROM &setY(int y){  
        y_ = y;  
        return *this;  
    }  
};
```

```
int main() {  
    CDROM cdrom;  
    cdrom.setX(1);  
    cdrom.setY(2);  
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda zwraca referencję do obiektu na którym została wykonana
  - **setX(...)** i **setY(...)** to settery

# wskaźnik **this**

```
class CDROM {  
    int x_;  
    int y_;  
    public:  
        CDROM &setX(int x){  
            x_ = x;  
            return *this;  
        }  
        CDROM &setY(int y){  
            y_ = y;  
            return *this;  
        }  
};
```

```
int main() {  
    CDROM cdrom;  
    cdrom.setX(1);  
    cdrom.setY(2);  
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda zwraca referencję do obiektu na którym została wykonana
  - **setX(...)** i **setY(...)** to settery
  - dodatkowo zwracają **referencję na typ klasy**

# wskaźnik **this**

```
class CDROM {  
    int x_;  
    int y_;  
    public:  
        CDROM &setX(int x){  
            x_ = x;  
            return *this;  
        }  
        CDROM &setY(int y){  
            y_ = y;  
            return *this;  
        }  
};
```

```
int main() {  
    CDROM cdrom;  
    cdrom.setX(1);  
    cdrom.setY(2);  
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda zwraca referencję do obiektu na którym została wykonana
  - **setX(...)** i **setY(...)** to settery
  - dodatkowo zwracają **referencję na typ klasy**
  - zwracają **referencję na obiekt na którym zostały wywołane**



# wskaźnik **this**

```
class CDROM {  
    int x_;  
    int y_;  
public:  
    CDROM &setX(int x){  
        x_ = x;  
        return *this;  
    }  
    CDROM &setY(int y){  
        y_ = y;  
        return *this;  
    }  
};
```

```
int main() {  
    CDROM cdrom;  
    cdrom.setX(1);  
    cdrom.setY(2);  
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda zwraca referencję do obiektu na którym została wykonana
  - setX(...) i setY(...) to settery
  - dodatkowo zwracają referencję na typ klasy
  - zwracają referencję na obiekt na którym zostały wywołane
  - Można wywołać settery “zwyczajnie”

# wskaźnik **this**

```
class CDROM {  
    int x_;  
    int y_;  
public:  
    CDROM &setX(int x){  
        x_ = x;  
        return *this;  
    }  
    CDROM &setY(int y){  
        y_ = y;  
        return *this;  
    }  
};  
  
int main() {  
    CDROM cdrom;  
    cdrom.setX(1).setY(2);  
}
```

- » Rozwiązywanie niejednoznacznych sytuacji
- » Przekazanie uchwytu obiektu do funkcji (poza klasę)
- » Metoda zwraca referencję do obiektu na którym została wykonana
  - setX(...) i setY(...) to settery
  - dodatkowo zwracają referencję na typ klasy
  - zwracają referencję na obiekt na którym zostały wywołane
  - Można wywołać settery “zwyczajnie”
  - Można użyć wywołania łańcuchowego



quiz

**PI10\_004**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

Dziękuję