Maciej Sobieraj

Wykład 6

Outline

1. **Dziedziczenie**
    1. Definiowanie hierarchii klas
    2. Klasy, dziedziczenie i zgodność typów
    3. Polimorfizm i metody wirtualne
    4. Obiekty jako parametry i dynamiczne rzutowanie
    5. **Różne dodatki**
    6. Słowo kluczowe *const*
    7. Przyjaźń w świecie “C++”

Więcej o konstruktorach kopiujących

- **Konstruktor kopiujący** jest specyficznym rodzajem konstruktora przeznaczonym do tworzenia bardziej lub mniej dokładnej kopii obiektu. Możesz rozpoznać ten konstruktor po **określonym nagłówku**.
- Zakładając, że klasa nazywa się A, jej konstruktor kopiowania zostanie zadeklarowany jako:
  - A(A &)
- **Niejawny konstruktor po prostu kopiuje** (
  po bicie) **obiekt źródłowy**

Więcej o konstruktorach kopiujących

```
#include <iostream>
using namespace std;
class Class {
      int data;
public:
      Class(int value) : data(value) {}
      void increment(void) {      data++; }
      int value(void) { return data; }
};
int main(void) {
      Class o1(123);
      Class o2 = o1;
      Class o3(o2);

      o1.increment();
      cout << o1.value() << endl;
      cout << o2.value() << endl;
      cout << o3.value() << endl;

      return 0;
}
```

Więcej o konstruktorach kopiujących

- Program wyświetli następujący wynik:
  - 124
  - 123
  - 123

Więcej o konstruktorach kopiujących

```
#include <iostream>
using namespace std;
class Class {
	int *data;
public:
	Class(int value) {
		data = new int;
		*data = value;
	}
	void increment(void) { (*data)++; }
	int value(void) { return *data; }
};
int main(void) {
	Class o1(123);
	Class o2 = o1;
	Class o3(o2);
	o1.increment();
	cout << o1.value() << endl;
	cout << o2.value() << endl;
	cout << o3.value() << endl;
	return 0;
}
```

Więcej o konstruktorach kopiujących

- Program wyświetli następujący wynik:
  - 124
  - 124
  - 124

Więcej o konstruktorach kopiujących

```
include <iostream>
using namespace std;
class Class {
	int *data;
public:
	Class(int value) {
		data = new int;
		*data = value;
	}
	void increment(void) {	(*data)++; }
	int value(void) { return *data; }
};

int main(void) {
	Class o1(123);
	Class o2(o1.value());
	Class o3(o2.value());
	o1.increment();
	cout << o1.value() << endl;
	cout << o2.value() << endl;
	cout << o3.value() << endl;
	return 0;
}
```

Więcej o konstruktorach kopiujących

- Program wyświetli następujący wynik:
  - 124
  - 123
  - 123

Więcej o konstruktorach kopiujących

```
#include <iostream>
using namespace std;
class Class {
	int *data;
public:
	Class(int value) {
		data = new int;
		*data = value;
	}
	Class(Class &source) {
		data = new int;
		*data = source.value();
	}
	void increment(void) {	(*data)++; }
	int value(void) { return *data; }
};
int main(void) {
	Class o1(123);
	Class o2 = o1;
	Class o3(o2);
	o1.increment();
	cout << o1.value() << endl;
	cout << o2.value() << endl;
	cout << o3.value() << endl;
	return 0;
}
```

Więcej o konstruktorach kopiujących

- Program wyświetli następujący wynik :
  - 124
  - 123
  - 123

Więcej o konstruktorach kopiujących

- Mechanizm przekazywania parametrów przez wartość zakłada, że funkcja działa na kopii rzeczywistego parametru.
- Jest to jasne, gdy rozważymy parametry typów prostych (jak ***int*** lub ***float***), ale staje się bardziej skomplikowane, gdy parametr jest obiektem.

Więcej o konstruktorach kopiujących

```
#include <iostream>
using namespace std;
class Dummy {
public:
	Dummy(int value) {}
	Dummy(Dummy &source) {
		cout << "Hi from the copy constructor!" << endl;
	}
};
void DoSomething(Dummy ob) {
	cout << "I'm here!" << endl;
}
int main(void) {
	Dummy o1(123);
	DoSomething(o1);
	return 0;
}
```

Więcej o konstruktorach kopiujących

- Program wyświetli następujący wynik:
  - Hi from the copy constructor!
  - I'm here!

Więcej o konstruktorach kopiujących

- Program spowoduje wyświetlenie co najmniej dwóch błędów (prywatny konstruktor kopiujący)

```
#include <iostream>
using namespace std;
class Dummy {
private:
	Dummy(Dummy &source) {}
public:
	Dummy(int value) {}
};
void DoSomething(Dummy ob) {
	cout << "I'm here!" << endl;
}
int main(void) {
	Dummy o1(123);
	Dummy o2 = o1;
	DoSomething(o1);
	return 0;
}
```

Więcej o konstruktorach domyślnych

- Klasa będzie niejawnie wyposażona w tak zwany domyślny **niejawny konstruktor (bez parametrów)**, ale konstruktor nic nie zrobi.

Więcej o konstruktorach domyślnych

```
#include <iostream>
using namespace std;
class NoConstructorsAtAll {
public:
	int i;
	float f;
	void Display(void) { cout << "i=" << i << ",f=" << f << endl; }
};
int main(void) {
	NoConstructorsAtAll  o1;
	NoConstructorsAtAll *o2;
	o2 = new NoConstructorsAtAll;
	o1.Display();
	o2 -> Display();
	return 0;
}
```

Więcej o konstruktorach domyślnych

- Klasa nie ma konstruktora. W rezultacie jej pola nie zostaną w żaden sposób zainicjowane. Wartości generowane przez metodę ***display*** są całkowicie losowe. The class has no constructor.
- Jeden z przykładowych wyników działania programu może być następujący:
  - i=2147344384,f=1.54143e-044
  - i=5641768,f=7.89812e-039

Więcej o konstruktorach domyślnych

- Domyślny konstruktor musi być niejawnie wywoływany podczas tworzenia nowego obiektu (dwa razy w poniższym przykładzie). We get error (twice)

```
#include <iostream>
using namespace std;
class WithConstructor {
public:
	int i;
	float f;
	WithConstructor(int a, float b) : i(a), f(b) { }
	void Display(void) { cout << "i=" << i << ",f=" << f << endl; }
};
int main(void) {
	WithConstructor  o1;
	WithConstructor *o2;
	o2 = new WithConstructor;
	o1.Display();
	o2 -> Display();
	return 0;
}
```

Więcej o konstruktorach domyślnych

- Chcąc naprawić problem powinniśmy zdefiniować konstruktor bez parametrów:
  - WithConstructor(void) : i(0), f(0.0) { }

```
#include <iostream>
using namespace std;
class WithConstructor {
public:
	int i;
	float f;
	WithConstructor(int a, float b) : i(a), f(b) { }
	void Display(void) { cout << "i=" << i << ",f=" << f << endl; }
};
int main(void) {
	WithConstructor  o1;
	WithConstructor *o2;
	o2 = new WithConstructor;
	o1.Display();
	o2 -> Display();
	return 0;
}
```

Więcej o konstruktorach domyślnych

- Zmieniliśmy nagłówek istniejącego konstruktora, dodając wartości domyślne do obu parametrów.

```
#include <iostream>
using namespace std;
class WithConstructor {
public:
	int i;
	float f;
	WithConstructor(int a = 0, float b = 0) : i(a), f(b) { }
	void Display(void) { cout << "i=" << i << ",f=" << f << endl; }
};
int main(void) {
	WithConstructor  o1;
	WithConstructor *o2;
	o2 = new WithConstructor;
	o1.Display();
	o2 -> Display();
	return 0;
}
```

Więcej o konstruktorach domyślnych

- Program wyświetli następujący wynik:
  - i=0,f=0
  - i=0,f=0

Kompozycje vs. konstruktory

```
#include <iostream>
using namespace std;
class A {
public:
	void Do(void) { cout << "A is doing something" << endl; }
};
class B {
public:
	void Do(void) { cout << "B is doing something" << endl; }
};
class Compo {
public:
	A f1;
	B f2;
};
int main(void) {
	Compo  co;
	co.f1.Do();
	co.f2.Do();
	return 0;
}
```

Kompozycje vs. konstruktory

- Program wyświetli następujący wynik:
  - A is doing something
  - B is doing something

Kompozycje vs. konstruktory

```
#include <iostream>
using namespace std;
class A {
public:
      A(A &src) { cout << "copying A..." << endl; }
      A(void) { }
      void Do(void) { cout << "A is doing something" << endl; }
};
class B {
public:
      B(B &src) { cout << "copying B..." << endl; }
      B(void){ }
      void Do(void) { cout << "B is doing something" << endl; }
};
class Compo {
public:
      Compo(void) { } ;
      A f1;
      B f2;
};
int main(void) {
      Compo  co1;
      Compo  co2 = co1;
      co2.f1.Do();
      co2.f2.Do();
      return 0;
}
```

Kompozycje vs. konstruktory

- Program wyświetli następujący wynik:
  - copying A...
  - copying B...
  - A is doing something
  - B is doing something
-

Kompozycje vs. konstruktory

```
#include <iostream>
using namespace std;
class A {
public:
      A(A &src) { cout << "copying A..." << endl; }
      A(void) { }
      void Do(void) { cout << "A is doing something" << endl; }
};
class B {
public:
      B(B &src) { cout << "copying B..." << endl; }
      B(void){ }
      void Do(void) { cout << "B is doing something" << endl; }
};
class Compo {
public:
      Compo(Compo &src) { cout << "Copying Compo..." << endl; }
      Compo(void) { } ;
      A f1;
      B f2;
};
int main(void) {
      Compo  co1;
      Compo  co2 = co1;
      co2.f1.Do();
      co2.f2.Do();
      return 0;
}
```

Kompozycje vs. konstruktory

- Program wyświetli następujący wynik:
  - Copying Compo...
  - A is doing something
  - B is doing something
- Jawny konstruktor kopiujący (napisany przez nas) nie wywołał żadnego z konstruktorów kopiujących klas składowych.

Kompozycje vs. konstruktory

- Jednym ze sposobów jest dodanie takiej linii:
  - Compo(Compo &src) : f1(src.f1), f2(src.f2) { cout << "Copying Compo..." << endl; }
- zamiast
  - Compo(Compo &src) { cout << "Copying Compo..." << endl; }
- Rozwiązanie jest także poprawne. Zmodyfikowany program zachowuje się tak, jak chcemy, wyświetlając następujący wynik:
  - copying A...
  - copying B...
  - Copying Compo...
  - A is doing something
  - B is doing something

Outline

1. **Dziedziczenie**
    1. Definiowanie hierarchii klas
    2. Klasy, dziedziczenie i zgodność typów
    3. Polimorfizm i metody wirtualne
    4. Obiekty jako parametry i dynamiczne rzutowanie
    5. Różne dodatki
    6. **Słowo kluczowe *const***
    7. Przyjaźń w świecie “C++”

Słowo kluczowe *const*

```
const int size1 = 100;
int const size2 = 100;
```

- ***size1*** i ***size2*** są zmiennymi typu ***int*** i mają wartość 100.
- Obie zmienne zachowują się jak stałe (dokładniej, jako zmienne tylko do odczytu).
- Zwróć uwagę, że słowo kluczowe ***const*** znajduje się w różnych miejscach.
- Obie formy są dopuszczalne.

Słowo kluczowe *const*

- Kompilator chroni obie zmienne przed modyfikacją. Następujące dwa wiersze kodu spowodują błędy kompilacji:
  - size1++;
  - size2 = size1;
- Możesz użyć obu symboli (nazw stałych) wszędzie tam, gdzie możesz użyć literału lub wyrażenia składającego się z literałów, jak w tym przykładzie:
  - **const int** size = 100;
  - **int** buffer[**size**];

Słowo kluczowe *const*

- Zauważ, że **nie możesz zadeklarować stałej *const* bez inicjalizacji** (pomyśl o tym przez chwilę, a przekonasz się, że jest to oczywiste). Poniższy wiersz spowoduje błąd kompilacji:
  - **const int** size;

Stałe agregaty

- Agregaty (struktury i tablice, a także tablice struktur i struktury tablic itp.) Mogą być zadeklarowane jako stałe, chociaż efekty działania są nieco inne.

```
const int points[5] = {1, 2, 4, 8, 16};
const struct { int key; } data = { 10 };
```

Stałe agregaty

- ***points*** i ***data*** są zmiennymi tylko do odczytu i nie wolno ich modyfikować. Obie poniższe linie są błędne:
  - --points[2];
  - data.key = 0;
- Niektóre kompilatory "C ++" mogą uznać poniższy wiersz za niepoprawny:
  - **int** array[points[2] + data.key];
- ponieważ kompilator może nie być w stanie określić liczby elementów tablicy podczas kompilacji.

Stałe wskaźniki

- Wskaźniki mogą być również zadeklarowane jako stałe.

```
int arr[5] = {1, 2, 4, 8, 16};
int * const iptr = arr + 2;
char * const cptr = "Why?";
```

- Zarówno ***iptr***, jak i ***cptr*** nie mogą być modyfikowane. Oznacza to, że poniższe linie spowodują błędy kompilacji :
  - --iptr;
  - ++cptr;

Stałe wskaźniki

- Elementy wskazywane przez wskaźniki *const* mogą być modyfikowane bez żadnych ograniczeń. Następujące dwie linie zostaną zaakceptowane i pomyślnie wykonane :
  - *iptr = 0;
  - *cptr = 'T';

Wskaźniki do stałych

- **Stałe wskaźniki nie są odpowiednikami dla wskaźników do stałych.**

```
int arr[5] = {1, 2, 4, 8, 16};
const int *iptr = arr + 2;
const char *cptr = "Why?";
```

- Słowa kluczowe const zmieniły lokalizację i teraz są umieszczone na początku deklaracji. Pamiętaj, że poniższy linie są również poprawne:
  - int const *iptr = arr + 2;
  - char const *cptr = "Why?";

Wskaźniki do stałych

- Zarówno ***iptr***, jak i ***cptr*** mogą być modyfikowane. Poniższe wiersze są poprawne :
  - --iptr;
  - ++cptr;
- Natomiast **elementów wskazanych przez te wskaźniki nie można już modyfikować**. Następujące dwie linie nie będą akceptowane:
  - *iptr = 0;
  - *cptr = 'T';

Stałe wskaźniki do stałych

- Oba powyższe warianty mogą być mieszane razem, dając stały wskaźnik do wartości stałej.

```
int arr[5] = {1, 2, 4, 8, 16};
const int * const iptr = arr + 2;
const char * const cptr = "Why?";
```

- Żadna z poniższych linii nie jest poprawna w odniesieniu do powyższej deklaracji:
  - --iptr;
  - ++cptr;
  - *iptr = 0;
  - *cptr = 'T';

Stałe parametry funkcji

- Dowolny z parametrów funkcji przekazywanych przez wartość może być zadeklarowany jako ***const***.

```
int fun(const int n) {
    return n * n;
}
```

- Zwróć uwagę, że efekty tych deklaracji są widoczne tylko wewnątrz funkcji i nie mają wpływu na świat zewnętrzny.
- Funkcja zwróci wynik ***n*n***

Stałe parametry funkcji

- **Dowolny z parametrów funkcji przekazywanych przez referencję może być zadeklarowany jako *const*.**
- Można powiedzieć, że jest to silniejsza forma poprzedniej deklaracji. Możemy to zrozumieć jako pewną **obietnicę** złożoną przez funkcję: *nie zamierzam modyfikować twojego aktualnego parametru.*

```
int fun(const int &n) {
        return n++;
}
```

- Ten fragment jest niepoprawny.

Wynik stałej funkcji

- Dowolna funkcja może zadeklarować swój wynik jako ***const***.

```
const char *fun(void) {
        return "Caution!";
}
```

- Ta linia zostanie odrzucona przez kompilator:
  - **char** *p = fun();
- Ta zostanie przyjęta:
  - **const char** *str = fun();

Zmienne stałe klasy

- Każda klasa może zadeklarować swoje pole jako ***const***.

```
class Class {
private:
    const int field;
public:
    Class(int n) : field(n) { };
    Class(Class &c) : field(0) { };
    Class(void) : field(1) { };
};
```

- **Pole klasy *const* musi zostać zainicjowane na liście inicjalizacji wewnątrz dowolnego z konstruktorów klas**. Każde inne przypisanie zostanie odrzucone.

Zmienne stałe klasy

- Wszystkie konstruktory inicjalizują pole ***const*** z inną wartością. Wszystkie inicjalizacje są prawidłowe.
- Poniższe fragmenty, wstawione do publicznej części Klasy, zostaną uznane za nieprawidłowe:
  - Class(**double** f) { field = f; }
  - **void** fun(**int** n) { field += n; }

Stałe obiekty

- Obiekt dowolnej klasy może być zadeklarowany jako ***const***.

```
class Class {
public:
    int field;
    Class(int n) : field(n) { };
    Class(Class &c) : field(0) { };
    Class(void) : field(1) { };
    void set(int n) { field = n; }
    int get(void) { return field; }
};
```

Stałe obiekty

- Załóżmy, że mamy następujące deklaracje (wszystkie poprawne):
  - Class o1(1);
  - **const** Class o2(2);
  - **int** i;
- Następujące trzy linie zostaną odrzucone:
  - o2.field = 3;
  - o2.set(1);
  - i = o2.get();
- Zostaną uznane za ważne, jeśli zastąpimy "o2" przez "o1".

Stałe funkcje klasy

- Każda z funkcji składowych klasy może zadeklarować się jako **stała**.
  Składnia deklaracji może być zaskakująca, ponieważ słowo kluczowe ***const*** jest umieszczone po liście parametrów:
  - type name(parameters) const; w deklaracji
  - type name(parameters) const { … } w definicji
- Jest to obietnica, że funkcja nie zmieni stanu obiektu

Stałe funkcje klasy

```
class Class {
public:
        int field;
        Class(int n) : field(n) { };
        Class(Class &c) : field(0) { };
        Class(void) : field(1) { };
        void set(int n) { field = n; }
        int get(void) const { return field; }
};
```

- W efekcie poniższy wiersz zostanie uznany za prawidłowy:
  - i = o2.get();

Outline

1. **Dziedziczenie**
    1. Definiowanie hierarchii klas
    2. Klasy, dziedziczenie i zgodność typów
    3. Polimorfizm i metody wirtualne
    4. Obiekty jako parametry i dynamiczne rzutowanie
    5. Różne dodatki
    6. Słowo kluczowe *const*
    7. **Przyjaźń w świecie “C++”**

Przyjaciel czy wróg?

- Przyjacielem klasy może być:
  - klasa (nazywana jest **friend class**)
  - funkcja (nazywa się to **friend function**)
- Przyjaciel (klasa lub funkcja) może uzyskać dostęp do tych składników ukrytych przed innymi. Przyjaciele mają prawo dostępu do lub korzystania z prywatnych i chronionych komponentów klasy.

Przyjaciel czy wróg?

```
#include <iostream>
using namespace std;
class Class {
friend class Friend;
private:
	int field;
	void print(void) { cout << "It's a secret, that field = " << field << endl; }
};
class Friend {
public:
	void DoIt(Class &c) { c.field = 100; c.print(); }
};
int main(void) {
	Class o;
	Friend f;

	f.DoIt(o);
	return 0;
}
```

Przyjaciel czy wróg?

- Zwróć uwagę, że **nie ma znaczenia, gdzie dodajesz deklarację przyjaźni**, tj. linię zaczynającą się od frazy:
  - friend class … ;
- może istnieć w dowolnej z części klasy (publicznej, prywatnej lub chronionej), ale musi być umieszczona poza jakąkolwiek funkcją lub agregatem.
- Program wyświetli wynik:
  - It's a secret, that field = 100

Zasady

- Istnieją pewne dodatkowe zasady, które należy wziąć pod uwagę:
  - klasa może **być przyjacielem wielu klas**
  - klasa może **mieć wielu przyjaciół**
  - przyjaciel przyjaciela **nie jest moim przyjacielem**
  - przyjaźń **nie jest dziedziczona** - podklasa musi zdefiniować własne przyjaźnie

Zasady

```
#include <iostream>
using namespace std;
class A {
friend class B;
friend class C;
private:
    int field;
protected:
    void print(void) { cout << "It's a secret, that field = " << field << endl; }
};
class C {
public:
    void DoIt(A &a) { a.print(); }
};
class B {
public:
    void DoIt(A &a, C &c) { a.field = 111; c.DoIt(a); }
};
int main(void) {
    A a; B b; C c;

    b.DoIt(a,c);
    return 0;
}
```

Zasady

- It's a secret, that field = 111

Funkcje zaprzyjaźnione

- **Funkcja może być również przyjacielem klasy**.
- Zasady są nieco inne niż wcześniej:
  - deklaracja przyjaźni musi zawierać **kompletny prototyp funkcji zaprzyjaźnionej** (łącznie z typem zwracanego wyniku); funkcja o tej samej nazwie, ale niekompatybilna w sensie zgodności parametrów, nie zostanie uznana za przyjaciela
  - klasa **może mieć wiele funkcji zaprzyjaźnionych**
  - funkcja **może być przyjacielem wielu klas**
  - klasa może uznać za przyjaciół zarówno funkcje globalne, jak i członkowskie

Funkcje zaprzyjaźnione

```
#include <iostream>
using namespace std;
class A;
class C {
public:
	void dec(A &a);
};
class A {
friend class B;
friend void C::dec(A&);
friend void DoIt(A&);
private:
	int field;
protected:
	void print(void) { cout << "It's a secret, that field = " << field << endl; }
};
void C::dec(A &a) { a.field--; }
class B {
public:
	void DoIt(A &a) { a.print(); }
};
void DoIt(A &a) {
	a.field = 99;
}
int main(void) {
	A a; B b; C c;

	DoIt(a);
	b.DoIt(a);
	return 0;
}
```

Funkcje zaprzyjaźnione

- Program wyświetli wynik:
  - It's a secret, that field = 99