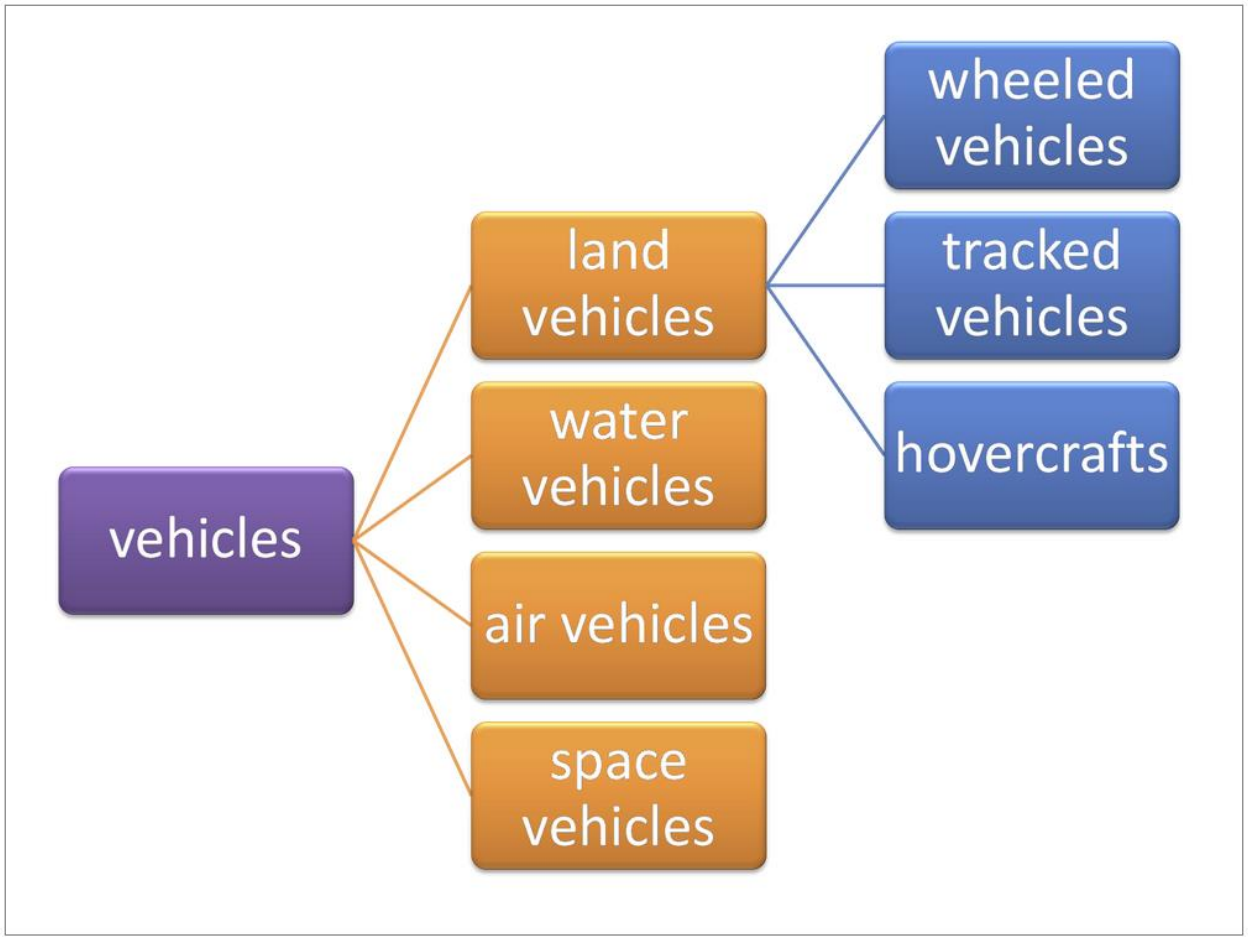
Maciej Sobieraj

Wykład 4

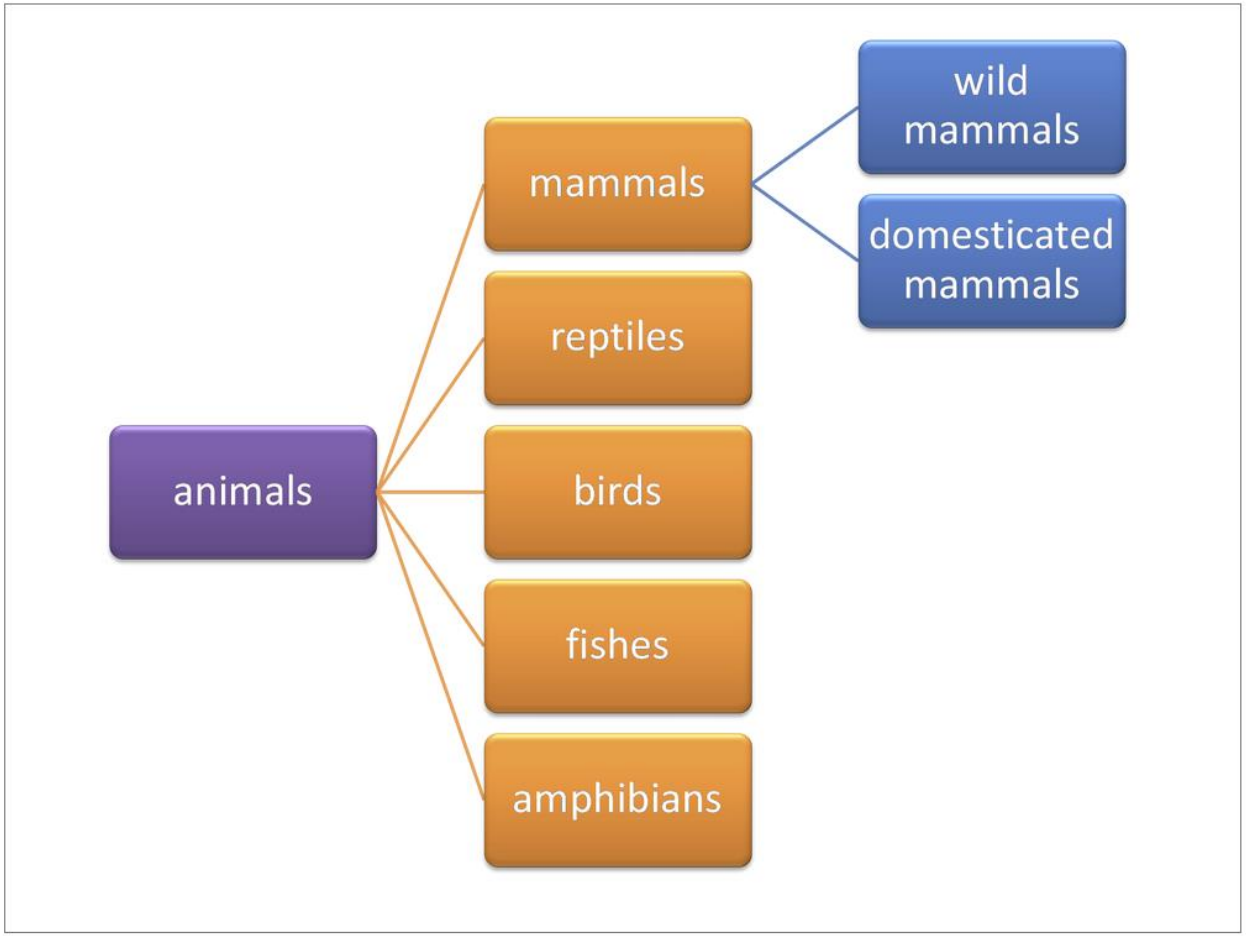
Outline

1. **Podstawy programwoania obiektowego**

Klasa – Co to?


vehicles
land vehicles
water vehicles
air vehicles
space vehicles
wheeled vehicles
tracked vehicles
hovercrafts

Klasa – Co to?


animals
mammals
wild mammals
domesticated mammals
reptiles
birds
fishes
amphibians

Klasa – Co to?

- Klasa to zbiór obiektów.
- Obiekt to byt należący do klasy.
- Obiekt jest wcieleniem wymagań i cech przypisanych do konkretnej klasy.
- Na przykład: dowolny samochód osobowy jest obiektem należącym do klasy „pojazdy”.

Co ma dowolny obiekt?

- Konwencja programowania obiektowego zakłada, że każdy istniejący obiekt może być wyposażony w trzy grupy atrybutów :
  - obiekt ma nazwę, która jednoznacznie identyfikuje go w jego macierzystej przestrzeni nazw (chociaż mogą istnieć również anonimowe obiekty)
  - obiekt ma zestaw indywidualnych właściwości, które czynią go oryginalnym, unikalnym lub wybitnym (chociaż istnieje możliwość, że niektóre obiekty mogą nie mieć żadnych właściwości)
  - obiekt ma zestaw zdolności do wykonywania określonych czynności, które mogą zmienić sam obiekt lub niektóre inne obiekty

Co ma dowolny obiekt?

- Oto dwie przykładowe frazy, które są dobrymi przykładami:
  - “Max is a large cat who sleeps all day”
    - Object name = Max
    - Home class = Cat
    - Property = Size (large)
    - Activity = Sleep (all day)
  - “A pink Cadillac went quickly”
    - Object name = Cadillac
    - Home class = Wheeled vehicles
    - Property = Colour (pink)
    - Activity = Drive (quickly)

Po co to wszystko?

- Programowanie obiektowe jest sztuką definiowania i rozszerzania klas.
- Klasa jest modelem bardzo specyficznej części rzeczywistości odzwierciedlającej właściwości i działania występujące w realnym świecie.

```
class OurClass { };
```

Pierwszy obiekt

- Nowo zdefiniowana klasa staje się odpowiednikiem typu i możemy jej użyć jako nazwy typu.
  Wyobraź sobie, że chcemy stworzyć jeden obiekt klasy ***OurClass***.

```
OurClass our_object;
```

Stos LIFO

- Stos to struktura opracowana do przechowywania danych w bardzo specyficzny sposób.
  - Alternatywną nazwą stosu jest **LIFO**. Jest to skrót opisujący zachowanie stosu: **"Last In - First Out"**.
- Zakładamy również, że element o indeksie 0 znajduje się na dole stosu.

```
int stack[100];
```

Wskaźnik stosu

- Potrzebujemy zmiennej, która może być odpowiedzialna za przechowywanie **liczby elementów aktualnie przechowywanych na stosie**.

```
int SP = 0;
```

Push

- Jesteśmy teraz gotowi do zdefiniowania funkcji, która **umieszcza** wartość na stosie.

```
void push(int value) {
    stack[SP++] = value;
}
```

Pop

- Teraz nadszedł czas, aby funkcja **usunęła** wartość ze stosu.

```
int pop(void) {
    return stack[--SP];
}
```

Stos w akcji

```
#include <iostream>

using namespace std;

int stack[100];
int SP = 0;

void push(int value) {
	stack[SP++] = value;
}

int pop(void) {
	return stack[--SP];
}

int main(void) {
	push(3);
	push(2);
	push(1);
	cout << pop() << endl;
	cout << pop() << endl;
	cout << pop() << endl;
	return 0;
}
```

Stos od zera

```
class Stack {
    int stackstore[100];
    int SP;
  };
```

Stos od zera

- Tego rodzaju dane są nazywane prywatnymi w programowaniu obiektowym. Prywatny oznacza, że tylko sama klasa może uzyskać do niego dostęp i go zmodyfikować.
- Jeśli chcemy oznaczyć pewną część danych klasy jako prywatne, musimy dodać słowo kluczowe przed deklaracjami.

```
class Stack {
private:
   int stackstore[100];
   int SP;
};
```

Stos od zera

- Teraz nadszedł czas na dwie funkcje, które realizują operacje **push** i **pop**. Język "C ++" zakłada, że funkcja tego rodzaju może być opisana na dwa różne sposoby:
  - **wewnątrz klasy**, gdy treść klasy zawiera pełną implementację metody
  - **poza klasą**, gdy treść zawiera tylko nagłówek funkcji; ciało funkcji jest umieszczone poza klasą

Stos od zera

- Chcemy wywoływać funkcje do dodawania i usuwania wartości ze stosu. Oznacza to, że obie funkcje powinny być dostępne dla każdego użytkownika klasy. W tym celu musimy użyć słowa kluczowego "**public**", aby podkreślić ten fakt.

```
class Stack {
private:
    int stackstore[100];
    int SP;
public:
    void push(int value);
    int pop(void) {
        return stackstore[--SP];
    }
};
```

Stos od zera

- Funkcje umieszczone poza ciałem klasy muszą być opisane w bardzo specyficzny sposób. Ich nazwy powinny być kwalifikowane przy użyciu nazwy klasy podstawowej i operatora "**::**".

```
class Stack {
   private:
      int stackstore[100];
      int SP;
   public:
      void push(int value);
      int pop(void) {
        return stackstore[--SP];
      }
};

void Stack::push(int value) {
      stackstore[SP++] = value;
}
```

Stos od zera

- Wyspecjalizowana funkcja wywoływana niejawnie za każdym razem, gdy tworzony jest nowy stos.
  - "Konstruktor" jest odpowiedzialny za prawidłową konstrukcję każdego nowego obiektu klasy.

```
class Stack {
  private:
    int stackstore[100];
    int SP;
  public:
    Stack(void) { SP = 0; }
    void push(int value);
    int pop(void) {
        return stackstore[--SP];
    }
};

void Stack::push(int value) {
    stackstore[SP++] = value;
}
```

Stos od zera

- Zwróć uwagę, w jaki sposób wywołujemy funkcję za pomocą obiektu.
- Jest to ta sama konwencja, którą już używaliśmy w łańcuchach znaków.

```
#include <iostream>

using namespace std;

int main(void) {
    Stack little_stack, another_stack, funny_stack;

    little_stack.push(1);
    another_stack.push(little_stack.pop() + 1);
    funny_stack.push(another_stack.pop() + 2);
    cout << funny_stack.pop() << endl;
    return 0;
}
```

Stos od zera

- Chcemy **nowego stosu z nowymi możliwościami**.
- Innymi słowy, chcemy skonstruować **podklasę** klasy Stack.

```
class AddingStack : Stack {
};
```

- Każdy obiekt klasy ***AddingStack*** może robić wszystko, co robi każdy obiekt klasy ***Stack***.

Stos od zera

- Zaczniemy od implementacji funkcji **push**. Tego właśnie od niej oczekujemy:
  - Dodawanie wartości do zmiennej ***sum***
  - wstawianie wartości na stos

```
class AddingStack : Stack {
   private:
      int sum;
   public:
      void push(int value);
      int pop(void);
};
```

Stos od zera

- Zaczniemy od implementacji funkcji **push**. Tego właśnie od niej oczekujemy:
  - Dodawanie wartości do zmiennej ***sum***
  - wstawianie wartości na stos

```
void AddingStack::push(int value) {
        sum += value;
        Stack::push(value);
}
```

Stos od zera

```
int AddingStack::pop(void) {
        int val = Stack::pop():
        sum -= val;
        return val;
}
```

```
int AddingStack::getSum(void) {
        return sum;
}
```

Stos od zera

```
AddingStack::AddingStack(void) : Stack() {
        sum = 0;
}
```

- Zwróć uwagę na wyrażenie ": ***Stack()***". Jest to żądanie wywołania **konstruktora klasy bazowej** przed uruchomieniem bieżącego konstruktora.

Stos od zera

```
class AddingStack : Stack {
   private:
      int sum;
   public:
      AddingStack(void);
      void push(int value);
      int pop(void);
      int getSum(void);
};
AddingStack::AddingStack(void) : Stack() {
      sum = 0;
}
void AddingStack::push(int value) {
      sum += value;
      Stack::push(value);
}
int AddingStack::pop(void) {
      int val = Stack::pop();
      sum -= val;
      return val;
}
int AddingStack::getSum(void) {
      return sum;
}
```

Stos od zera

```
#include <iostream>

using namespace std;

int main(void) {
	AddingStack super_stack;

	for(int i = 1; i < 10; i++)
		super_stack.push(i);
	cout << super_stack.getSum() << endl;
	for(int i = 1; i < 10; i++)
		super_stack.pop();
	cout << super_stack.getSum() << endl;
	return 0;
}
```

Składniki klasy

- Ponieważ wszystkie komponenty są zadeklarowane bez użycia **specyfikatora dostępu** (wśród deklaracji nie dodano żadnego publicznego ani prywatnego słowa kluczowego) wszystkie trzy komponenty są **prywatne**. Oznacza to, że klasę zdefiniowaną w następujący sposób:

```
class A {
    Type Var;
};
```

- należy czytać jako:

```
class A {
    private:
    Type Var;
};
```

Specyfikatory dostępu

```
class Class {
        int value;
        void setVal(int value);
        int getVal(void);
};
```

- Klasa została przebudowana, aby pokazać wykorzystanie **specyfikatorów dostępu**.

```
class Class {
public:
        void setVal(int value);
        int getVal(void);
private:
        int value;
};
```

Tworzenie obiektu

Class the_object;

- Składniki o dostępie ***public*** są dostępne z poziomu obiektu:

  the_object.setVal(0);

- Składniki o dostępie ***private*** nie:

  ~~the_object.value = 0;~~

Nadpisywanie nazw komponentów

- Funkcja ***setVal*** używa parametru o nazwie ***value***.
- Parametr zastępuje komponent klasy o nazwie ***value***.

```
class Class {
public:
   void setVal(int value) {
         Class::value = value;
   }
   int getVal(void);
private:
   int value;
};
```

Wskaźnik "this"

- Zakładamy, że każdy obiekt jest wyposażony w specjalny komponent zawierający następujące cechy :
  - Jego nazwa to **this**
  - nie może być jawnie zadeklarowany (jest to słowo kluczowe), więc **nie może zostać nadpisany**
  - **jest wskaźnikiem do bieżącego obiektu** - każdy obiekt ma własną kopię tego wskaźnika

Wskaźnik "this"

- Ogólna zasada mówi, że:
  - jeśli ***S*** jest **strukturą** lub klasą, a ***S*** ma składnik o nazwie **C** i
  - jeśli ***p*** jest wskaźnikiem do struktury typu ***S***
  - to składnik ***C*** może być dostępny na dwa następujące sposoby:
    - (* p) .C // p jest jawnie określone w celu uzyskania dostępu do składnika C.
    - p-> C // p jest domyślnie określone w celu uzyskania dostępu do składnika C.

Wskaźnik "this"

```
class Class {
public:
    void setVal(int value) {
        this -> value = value;
    }
    int getVal(void);
private:
    int value;
};
```

Definiowanie nazw komponentów

- Jeśli jakiekolwiek ciało funkcji klasy jest określone poza ciałem klasy, to nazwa funkcji musi być zdefiniowana za pomocą nazwy klasy podstawowej i operatora "::"

```
class Class {
public:
   void setVal(int value) {
         this -> value = value;
   }
   int getVal(void);
private:
   int value;
};

int Class::getVal(void) {
    return value;
}
```

Definiowanie nazw komponentów

- Nazwy funkcji klas **mogą być przeciążone**, podobnie jak zwykłe nazwy funkcji.
  - Pierwsza ma jeden parametr i ustawia pole ***value*** jako wartość parametru.
    Druga nie ma parametrów i ustawia pole ***value*** na -2

```
class Class {
public:
        void setVal(int value) { this -> value = value;  }
        void setVal(void) { value = -2; }
        int getVal(void) { return value; }
private:
        int value;
};
```

Konstruktory

- **Funkcja o nazwie identycznej jak nazwa klasy macierzystej nazywa się konstruktorem**.
- Konstruktor ma na celu **skonstruowanie obiektu podczas jego tworzenia**, tj. inicjowanie wartości pól, przydzielanie pamięci, tworzenie innych obiektów itp.
  Konstruktor może uzyskać dostęp do wszystkich komponentów obiektu, jak każda inna funkcja klasy, ale nie powinien być wywoływany bezpośrednio.

Konstruktory

- Konstruktora nie można deklarować przy użyciu specyfikacji typu ***return***.
- Deklaracja obiektu w następujący sposób:
  - **Class** object;
- **Powoduje niejawne wywołanie konstruktora**.
- Uwaga - nie możesz tego robić:
  - ~~object.Class()~~
  - ~~Class::Class();~~

Konstruktory

```
class Class {
public:
	Class(void) { this -> value = -1; }
	void setVal(int value) { this -> value = value; }
	int getVal(void) { return value; }
private:
	int value;
};
```

- **Class** object;
- cout << object.getVal() << endl;

Przeciążanie nazw konstruktora

- Konstruktory mogą być również przeciążone, w zależności od konkretnych potrzeb i wymagań.

```
class Class {
public:
    Class(void) { this -> value = -1; }
    Class(int val) { this -> value = val; }
    void setVal(int value) { this -> value = value; }
    int getVal(void) { return value; }
private:
    int value;
};
```

  - Class object1, object2(100);
  - cout << object1.getVal() << endl;
  - cout << object2.getVal() << endl;

Przeciążanie nazw konstruktora

- Jeśli klasa ma konstruktor (a dokładniej co najmniej jeden konstruktor), jeden z nich musi zostać wybrany podczas tworzenia obiektu

```
class Class {
public:
        Class(int val) { this -> value = val; }
        void setVal(int value) { this -> value = value; }
        int getVal(void) { return value; }
private:
        int value;
};
```

  - Class object(2);
  - ~~Class object;~~

Konstruktory kopiujące

- Istnieje specjalny rodzaj konstruktora przeznaczony do **kopiowania jednego obiektu do drugiego**.
- Jeśli konstruktor kopiujący nie istnieje w danej klasie, a inicjator jest rzeczywiście używany podczas deklaracji obiektu, jego zawartość będzie faktycznie (w dosłownym tego słowa znaczeniu) skopiowana "pole po polu„
- Zauważ, że słowo kluczowe **const** użyte w deklaracji parametru jest **obietnicą**, że funkcja nie będzie próbowała modyfikować wartości przechowywanych w wywoływanym obiekcie.

Konstruktory kopiujące

```
#include <iostream>
using namespace std;
class Class1 {
public:
	Class1(int val) { this -> value = val; }
	Class1(Class1 const &source) { value = source.value + 100; }
	int value;
};
class Class2 {
public:
	Class2(int val) { this -> value = val; }
	int value;
};
int main(void) {
	Class1 object11(100), object12 = object11;
	Class2 object21(200), object22 = object21;
	cout << object12.value << endl;
	cout << object22.value << endl;
	return 0;
}
```

Wycieki pamięci

- Brak czyszczenia pamięci spowoduje zjawisko "wycieku pamięci", gdzie pamięć nieużywana (ale nadal przydzielona!) powiększa się, wpływając na wydajność systemu.
- Możemy sobie wyobrazić, że tworzenie obiektów składa się z dwóch faz:
  - sam obiekt jest tworzony, a część pamięci jest niejawnie przydzielona do obiektu
  - konstruktor jawnie przydziela inną część pamięci
- Niestety, pamięć jawnie przydzielona przez konstruktora **pozostaje przydzielona**.

Wycieki pamięci

```
#include <iostream>
using namespace std;
class Class {
public:
	Class(int val) {
		value = new int[val];
		cout << "Allocation (" << val << ") done." << endl;
	}
	int *value;
};
void MakeALeak(void) {
	Class object(1000);
}
int main(void) {
	MakeALeak();
	return 0;
}
```

Destruktory

- Możemy zabezpieczyć się przed tym niebezpieczeństwem, definiując specjalną funkcję zwaną **destruktorem**. Destruktory mają następujące ograniczenia:
  - jeśli klasa ma nazwę ***X***, jej destruktor ma nazwę ***~X***
  - klasa może mieć maksymalnie jeden destruktor
  - destruktor musi być funkcją bez parametrów
  - destruktor nie powinien być wywoływany jawnie

Destruktory

```
#include <iostream>
using namespace std;
class Class {
public:
	Class(int val) {
		value = new int[val];
		cout << "Allocation (" << val << ") done." << endl;
	}
	~Class(void) {
		delete [] value;
		cout << "Deletion done." << endl;
	}
	int *value;
};
void MakeALeak(void) {
	Class object(1000);
}
int main(void) {
	MakeALeak();
	return 0;
}
```

Słowo kluczowe “auto”

- Wszystkie zmienne w twoim kodzie należą do jednej z dwóch kategorii:
  - **zmienne automatyczne, tworzone i niszczone**, czasami wielokrotnie i **automatycznie** (stąd ich nazwa) podczas wykonywania programu
  - **zmienne statyczne, istniejące cały czas** podczas wykonywania całego programu
- Języki programowania "C" i "C ++" zakładają, że **wszystkie zmienne są domyślnie automatyczne**, chyba że są jawnie zadeklarowane jako statyczne.

Słowo kluczowe “auto”

```
#include <iostream>
using namespace std;
void fun(void) {
	auto int var = 99;
	cout << "var = " << ++var << endl;
}
int main(void) {
	for(int i = 0; i < 5; i++)
		fun();
	return 0;
}
```

Słowo kluczowe “auto”

- Zmienna ***var*** istnieje nawet wtedy, gdy funkcja ***fun*** nie działa, więc wartość zmiennej jest zachowywana pomiędzy kolejnymi wywołaniami funkcji ***fun***

```
#include <iostream>
using namespace std;
void fun(void) {
	static int var = 99;
	cout << "var = " << ++var << endl;
}
int main(void) {
	for(int i = 0; i < 5; i++)
		fun();
	return 0;
}
```

Instancje klasy

- Każdy obiekt utworzony z określonej klasy nazywa się **instancją** klasy.
- Żaden z tych komponentów nie istnieje, dopóki nie zostanie utworzona pierwsza instancja.

```
#include <iostream>
using namespace std;
class Class {
public:
	int val;
	void print(void) { cout << val << endl; }
};
int main(void) {
	Class::val = 0;
	Class::print();
	return 0;
	}
```

Instancje klasy

- Wszystkie reguły na poprzednim slajdzie są prawdziwe, jeśli odnoszą się do **niestatycznych** elementów klasy (zarówno pól, jak i funkcji).
- **Komponent statyczny istnieje przez całe życie programu**. Co więcej, zawsze **istnieje tylko jeden składnik**, niezależnie od liczby instancji klasy.
- Można powiedzieć, że wszystkie instancje mają te same statyczne komponenty.

Statyczne zmienne klasy

```
#include <iostream>
using namespace std;
class Class {
public:
	static int Static;
	int NonStatic;
	void print(void) {
		cout << "Static = " << ++Static <<
			", NonStatic = " << NonStatic << endl;
	}
};
int Class::Static = 0;
int main(void) {
	Class instance1, instance2;
	instance1.NonStatic = 10;
	instance2.NonStatic = 20;
	instance1.print();
	instance2.print();
	return 0;
	}
```

Statyczne zmienne klasy

- Ten program wyświetla następujące dane na ekranie:
  - Static = 1, NonStatic = 10
  - Static = 2, NonStatic = 20

Statyczne zmienne klasy

- Zauważ jeszcze raz, że do pola Counter można uzyskać bezpośredni dostęp, gdy jest on używany w klasie oraz poza klasą z operatorem "::". Możliwe jest również uzyskanie dostępu do zmiennej statycznej za pośrednictwem dowolnej z istniejących instancji klasy, na przykład:
  - cout << b.Counter ;

Statyczne zmienne klasy

```
#include <iostream>
using namespace std;
class Class {
public:
	static int Counter;
	Class(void) { ++Counter; };
	~Class(void) {
		--Counter;
		if(Counter == 0) cout << "Bye, bye!" << endl;
	};
	void HowMany(void) { cout << Counter << " instances" << endl; }
};
int Class::Counter = 0;
int main(void) {
	Class a;
	Class b;
	cout << Class::Counter << " instances so far" << endl;
	Class c;
	Class d;
	d.HowMany();
	return 0;
	}
```

Statyczne zmienne klasy

- Program wyświetli na ekranie następujący wynik:
  - 2 instances so far
  - 4 instances
  - Bye, bye!

Statyczne zmienne klasy

```
#include <iostream>
using namespace std;
class Class {
      static int Counter;
public:
      Class(void) {
            ++Counter;
      };
      ~Class(void) { --Counter; if(Counter == 0)
                        cout << "Bye, bye!" << endl;
      };
      void HowMany(void) { cout << Counter << " instances" << endl; }
};
int Class::Counter = 0;
int main(void) {
      Class a;
      Class b;
      b.HowMany();
      Class c;
      Class d;
      d.HowMany();
      return 0;
      }
```

Statyczne zmienne klasy

- Zauważ, że wszelkie próby uzyskania dostępu do zmiennej ***Counter*** wyrażone w ten sposób :
  - ~~Class::Counter = 1;~~
- są **zabronione**.

Statyczne zmienne klasy

- **Funkcja statyczna**, podobnie jak zmienna statyczna, może być również dostępna (lub bardziej precyzyjnie, wywoływana), gdy nie zostały utworzone żadne wystąpienia klasy..
- Zauważ, że funkcję statyczną można wywołać z wnętrza klasy:
  - HowMany();
- lub za pomocą dowolnej z istniejących instancji:
  - b.HowMany();

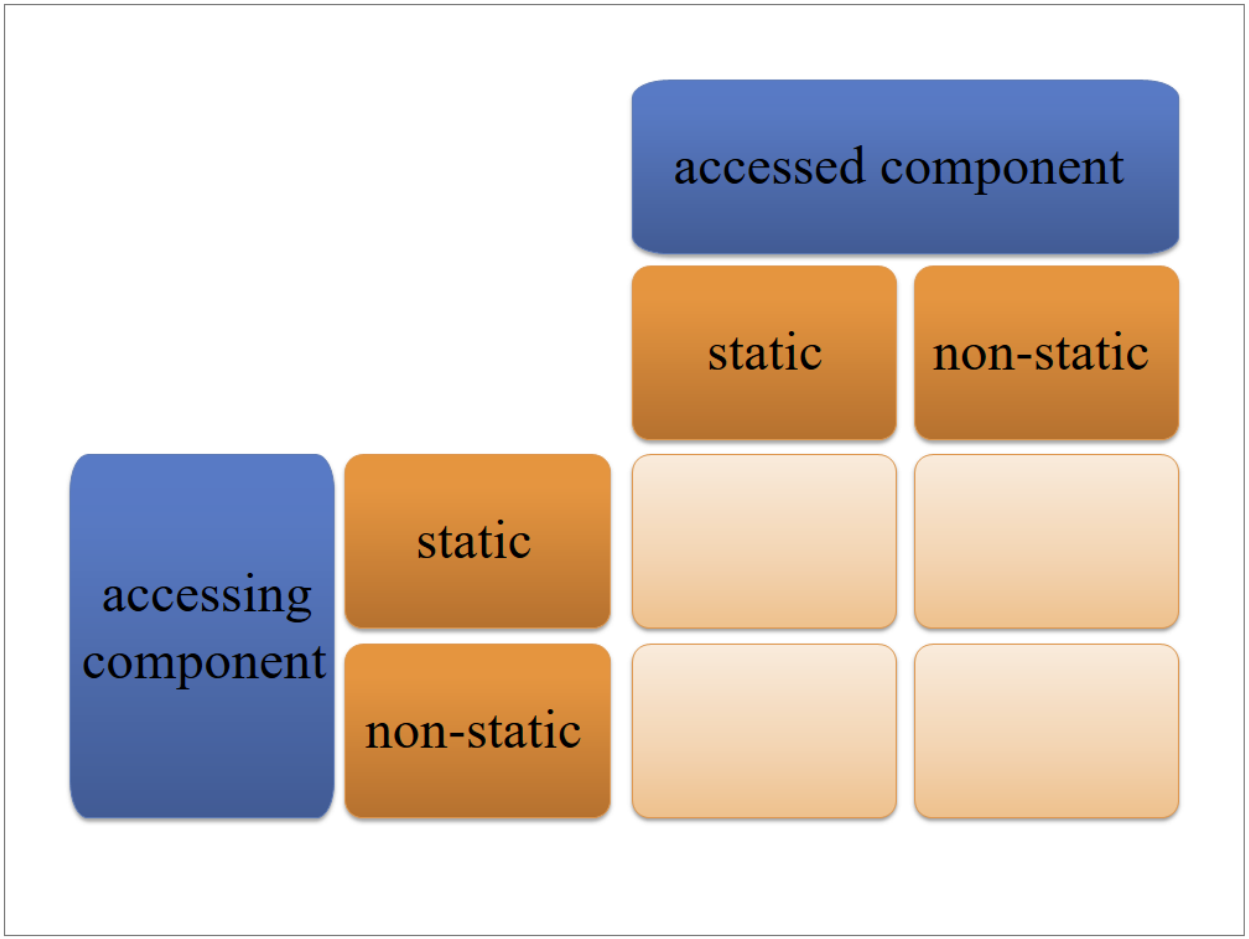
Statyczne zmienne klasy

```
#include <iostream>
using namespace std;
class Class {
	static int Counter;
public:
	Class(void) {
		++Counter;
	};
	~Class(void) {
		--Counter;
		if(Counter == 0)
			cout << "Bye, bye!" << endl;
	};
	static void HowMany(void) { cout << Counter << " instances" << endl; }
};
int Class::Counter = 0;
int main(void) {
	Class::HowMany();
	Class a;
	Class b;
	b.HowMany();
	Class c;
	Class d;
	d.HowMany();
	return 0;
	}
```

Statyczne a niestatyczne komponenty

- Współistnienie zarówno składników statycznych, jak i niestatycznych w obrębie jednej klasy powoduje dodatkowe problemy, które musimy wziąć pod uwagę. Możemy zdefiniować cztery konkretne zdarzenia, gdy oba typy komponentów wchodzą ze sobą w interakcje.
- Są to:
  - element **statyczny** uzyskuje dostęp do elementu **statycznego**
  - element **statyczny** uzyskuje dostęp do elementu **niestatycznego**
  - **niestatyczny** komponent uzyskuje dostęp do elementu **statycznego**
  - **niestatyczny** komponent uzyskuje dostęp do elementu **niestatycznego**

Statyczne a niestatyczne komponenty

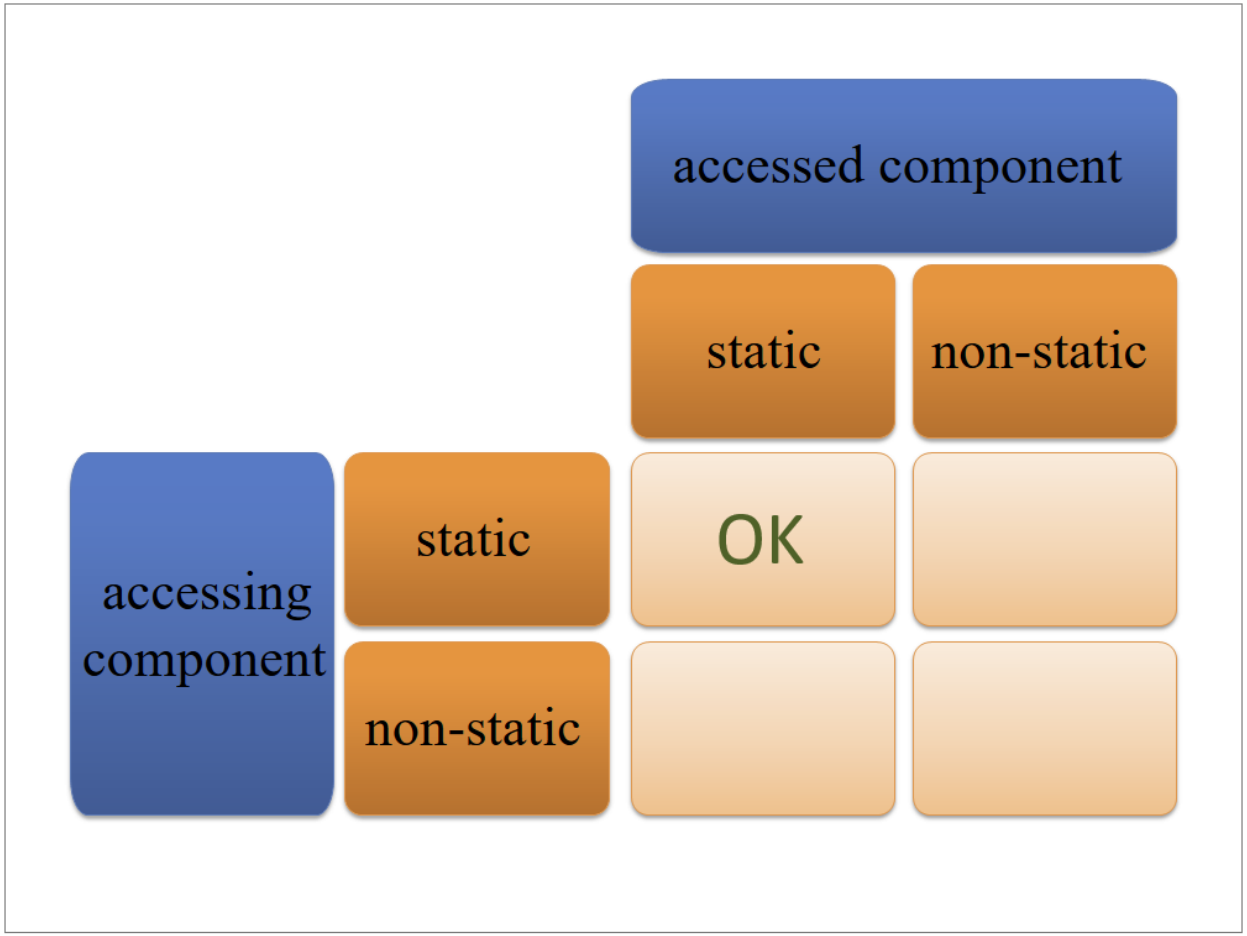


Interakcja static → static

- Pierwszy program testowy (pokazany tutaj →) demonstruje przypadek, gdy **statyczna funkcja o nazwie *funS2* próbuje wywołać inną statyczną funkcję o nazwie *funS1***.

```
#include <iostream>

using namespace std;

class Test {
public:
      static void funS1(void) { cout << "static" << endl; }
      static void funS2(void) { funS1(); }
};

int main(void) {
      Test object;
      Test::funS2();
      object.funS2();
      return 0;
}
```

Interakcja static → static



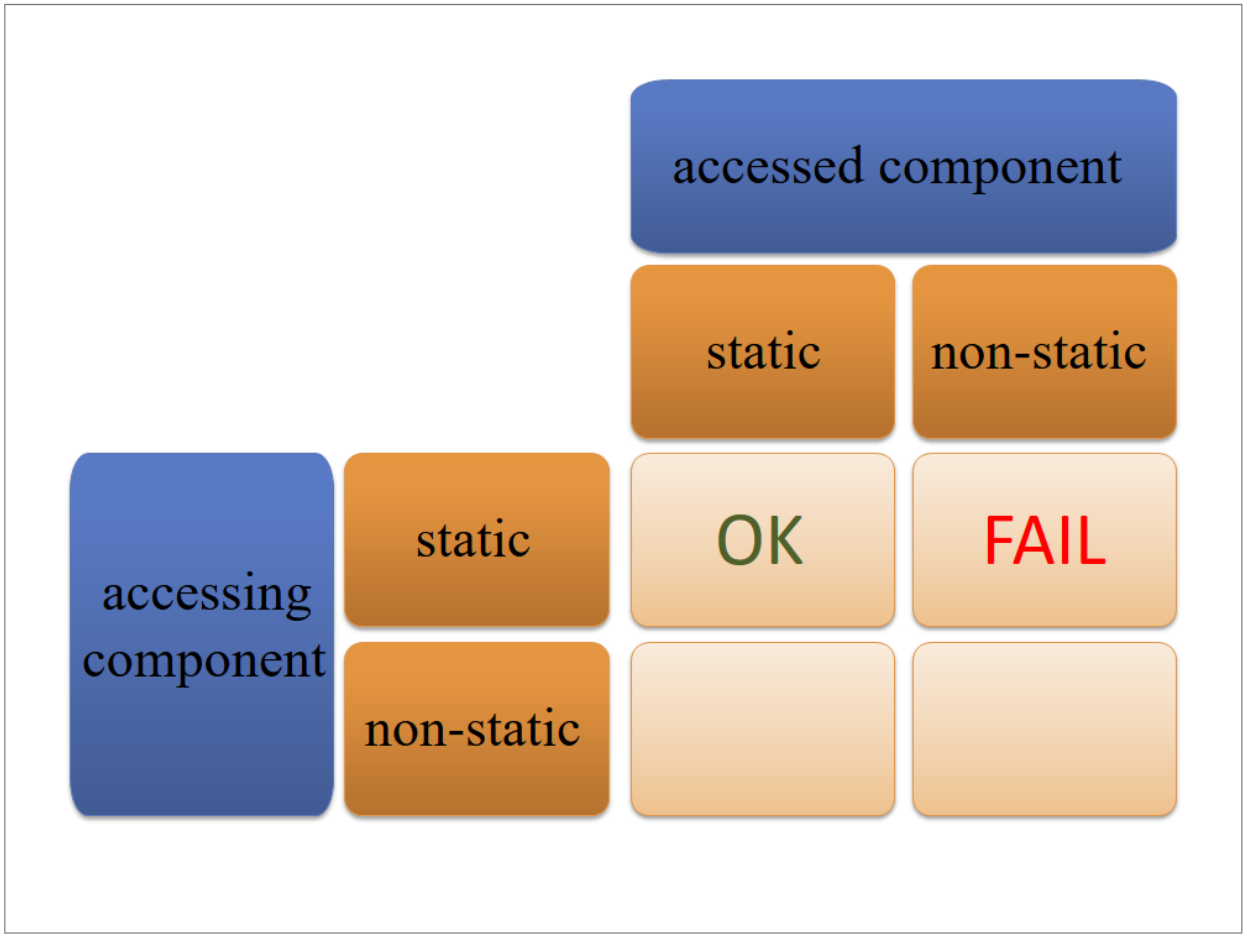
Interakcja static → non-static

- Drugi program testowy (pokazany tutaj →) demonstruje przypadek, gdy **statyczna funkcja o nazwie *funS1* próbuje wywołać niestatyczną funkcję o nazwie *funN1***.

```
#include <iostream>
using namespace std;
class Test {
public:
      void funN1(void) { cout << "non-static" << endl; }
      static void funS1(void) { funN1(); }
};
int main(void) {
      Test object;
      Test::funS1();
      object.funS1();
      return 0;
}
```

  - Tego programu **nie można pomyślnie skompilować**.

Interakcja static → non-static

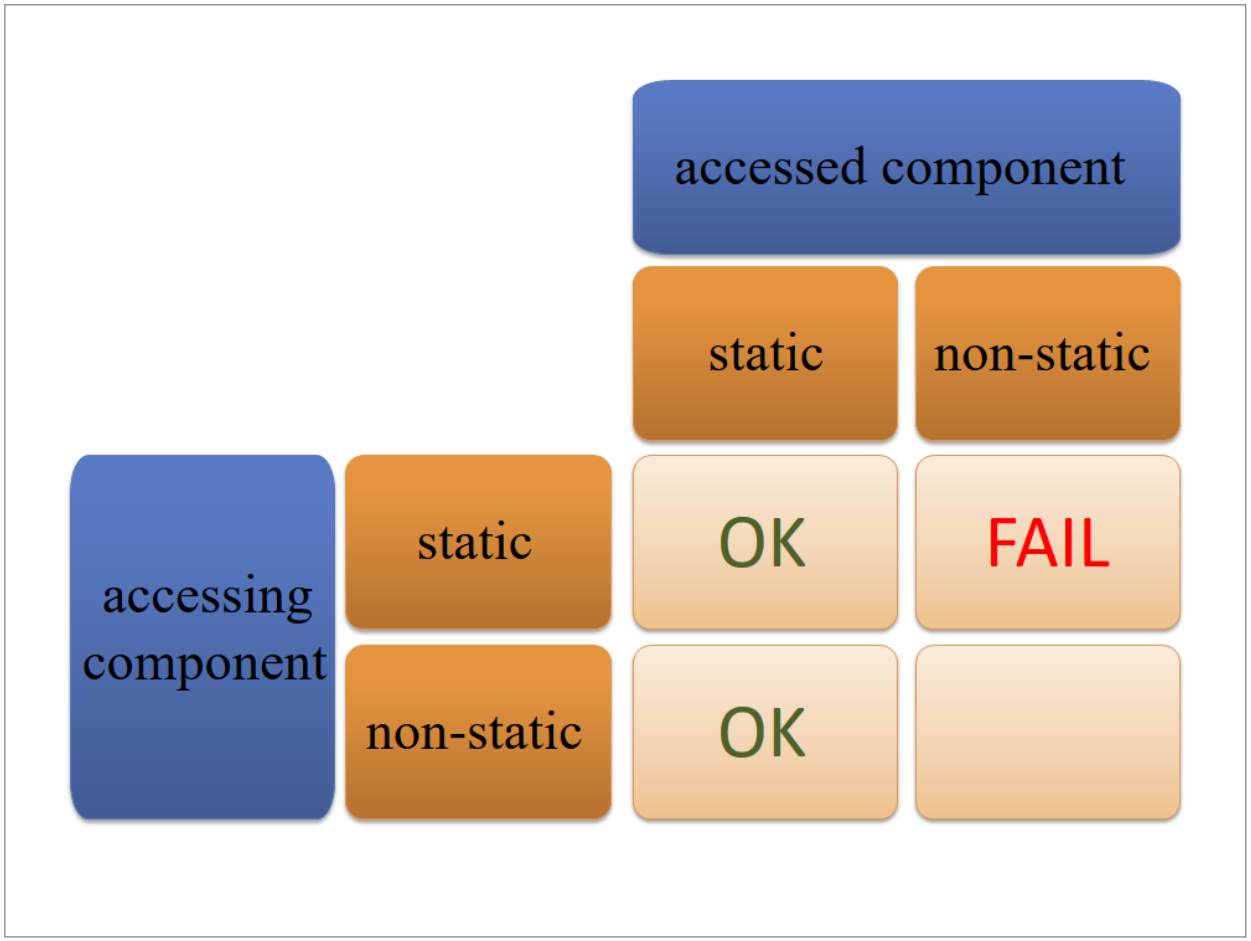


Interakcja nonstatic → static

- Trzeci przypadek testowy (można go znaleźć tutaj →) odnosi się do sytuacji, w której **niestatyczna funkcja o nazwie *funN1* wywołuje funkcję statyczną o nazwie *funS1***.

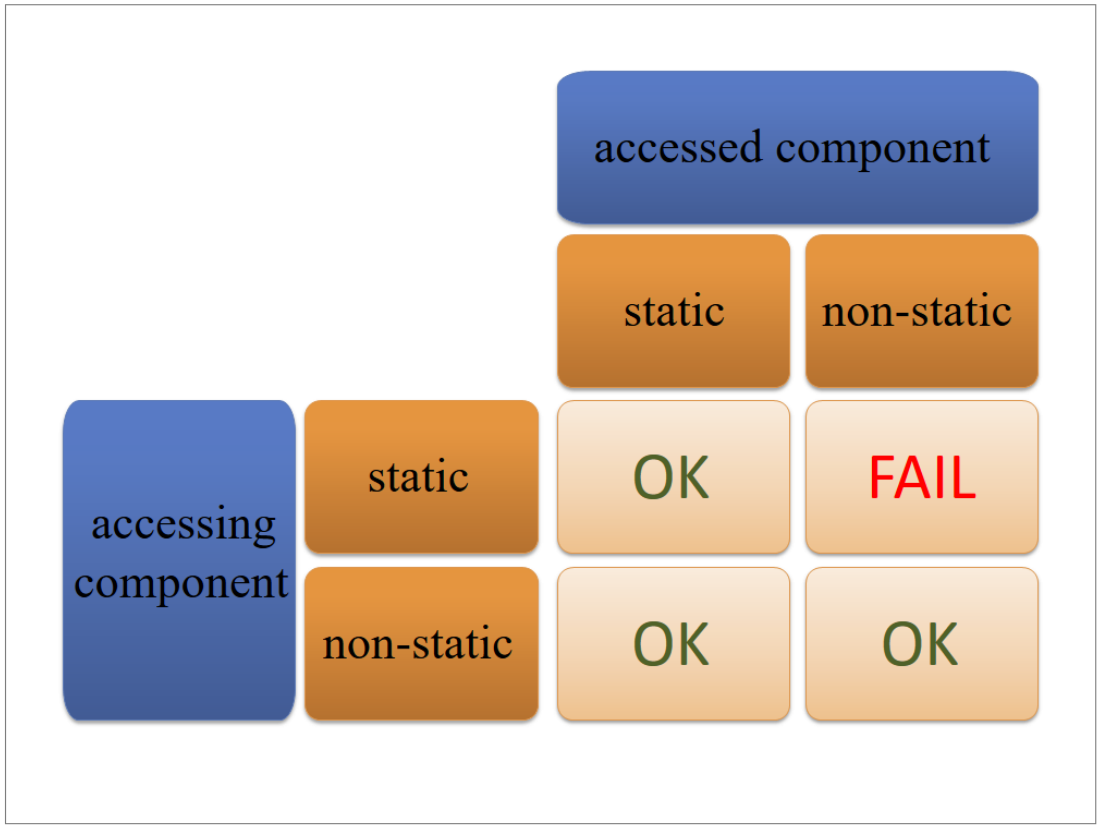
```
#include <iostream>

using namespace std;

class Test {
public:
	static void funS1(void) { cout << "static" << endl; }
	void funN1(void) { funS1(); }
};

int main(void) {
	Test object;
	object.funN1();
}
```

Interakcja nonstatic → static



Interakcja nonstatic → nonstatic

- Możliwe jest wywołanie funkcji niestatycznej poprzez funkcję niestatyczną



Wskaźniki do obiektów

- Obiekty mogą również istnieć jako dynamicznie tworzone i niszczone obiekty. Innymi słowy, **obiekty mogą pojawiać się na żądanie** - kiedy są potrzebne - i znikać w ten sam sposób.

Wskaźniki do obiektów

```
#include <iostream>
using namespace std;
class Class {
public:
	Class(void) {
		cout << "Object constructed!" << endl;
	}
	~Class(void) {
		cout << "Object destructed!" << endl;
}
};
int main(void) {
	Class *ptr;

	ptr = new Class();
	delete ptr;
	return 0;
	}
```

Wskaźniki do pól

- Wszystkie zmienne, w tym obiekty, ożywione w "zwykły" sposób (poprzez deklarację, a nie przez użycie słowa kluczowego ***new***), żyją w **oddzielnym obszarze pamięci zwanym stosem (stack)**. Jest to region pamięci przeznaczony do przechowywania wszystkich **automatycznych jednostek**.
- Utworzone elementy "na żądanie" (przy użyciu słowa kluczowego ***new***) są tworzone w określonym regionie pamięci, zwanym zwykle **stertą (heap).**

Wskaźniki do pól

- Obiekt przechowywany w stercie musi być dostępny w sposób przypominający dostęp do dynamicznie przydzielanych struktur.
  Nie wolno używać zwykłej notacji "kropkowanej", ponieważ nie ma struktury (obiektu), która może pełnić rolę lewego argumentu operatora ".",
  Chyba, że usuniesz wskaźnik.
- **Zamiast tego należy użyć operatora "strzałki" (->)**.

Wskaźniki do pól

```
#include <iostream>
using namespace std;
class Class {
public:
	Class(void) {
		cout << "Object constructed!" << endl;
	}
	~Class(void) {
		cout << "Object destructed!" << endl;
	}
	int value;
};
int main(void) {
	Class *ptr;

	ptr = new Class;
	ptr -> value = 0;
	cout << ++(ptr -> value) << endl;
	delete ptr;
	return 0;
}
```

Wskaźniki do funkcji

- Do funkcji danej klasy wywoływanych przez obiekt, do którego można uzyskać dostęp za pośrednictwem wskaźnika, należy uzyskać dostęp również za pomocą operatora strzałki

Wskaźniki do funkcji

```
#include <iostream>
using namespace std;
class Class {
public:
	Class(void) {
		cout << "Object constructed!" << endl;
	}
	~Class(void) {
		cout << "Object destructed!" << endl;
	}
	void IncAndPrint(void) {
		cout << "value = " << ++value << endl;
	}
	int value;
};
int main(void) {
	Class *ptr;

	ptr = new Class;
	ptr -> value = 1;
	ptr -> IncAndPrint();
	delete ptr;
	return 0;
}
```

Wybór konstruktora

- Jeśli klasa ma więcej niż jeden konstruktor, jeden z nich może zostać wybrany podczas tworzenia obiektu. Odbywa się to poprzez określenie formy listy parametrów powiązanej z nazwą klasy.
- Lista powinna być jednoznacznie zgodna z jednym z dostępnych konstruktorów klasy.

Wybór konstruktora

```
#include <iostream>
using namespace std;
class Class {
public:
	Class(void) { cout << "Object constructed (#1)" << endl; }
	Class(int v) { value = v; cout << "Object constructed (#2)" << endl; }
	~Class(void) { cout << "Object destructed! val = " << value << endl; }
	void IncAndPrint(void) {
		cout << "value = " << ++value << endl;
	}
	int value;
};
int main(void) {
	Class *ptr1, *ptr2;

	ptr1 = new Class;
	ptr2 = new Class(2);
	ptr1 -> value = 1;
	ptr1 -> IncAndPrint();
	ptr2 -> IncAndPrint();
	delete ptr2;
	delete ptr1;
	return 0;
}
```

Tablica wskaźników do obiektów

```
#include <iostream>
using namespace std;
class Array {
      int *values;
      int  size;
public:
      Array(int siz) {
            size = siz; values = new int[size];
            cout << "Array of " << size << " ints constructed." << endl;
      }
      ~Array(void) {
            delete [] values;
            cout << "Array of " << size << " ints destructed." << endl;

      }
      int Get(int ix) { return values[ix]; }
      void Put(int ix, int val) { values[ix] = val; }
};
int main(void) {
      Array *arr = new Array(2);

      for(int i = 0; i < 2; i++)
            arr->Put(i, i + 100);
      for(int i = 0; i < 2; i++)
            cout << "#" << i + 1 << ":" << arr->Get(i) << endl;
      delete arr;
      return 0;
}
```

Tablica wskaźników do obiektów

```
#include <iostream>
using namespace std;
class Array {
      int *values;
      int  size;
public:
      Array(int siz) {
            size = siz; values = new int[size];
            cout << "Array of " << size << " ints constructed." << endl;
      }
      ~Array(void) {
            delete [] values;
            cout << "Array of " << size << " ints destructed." << endl;

      }
      int Get(int ix) { return values[ix]; }
      void Put(int ix, int val) { values[ix] = val; }
};
int main(void) {
      Array *arr[2] = { new Array(2), new Array(2) };

      for(int i = 0; i < 2; i++)
            for(int j = 0; j < 2; j++)
                  arr[i]->Put(j, j + 10 + i);
      for(int i = 0; i < 2; i++) {
            for(int j = 0; j < 2; j++)
                  cout << "#" << i + 1 << ":" << arr[i]->Get(j) << "; ";
            cout << endl;
      }
      delete arr[0];delete arr[1];
      return 0;
}
```

Obiekty w obiektach

- Obiekt dowolnej klasy może być polem obiektu dowolnej innej klasy.

```
#include <iostream>
using namespace std;
class Element {
	int value;
public:
	int Get(void) { return value; }
	void Put(int val) { value = val; }
};
class Collection {
	Element el1, el2;
public:
	int Get(int elno) { return elno == 1 ? el1.Get() : el2.Get(); }
	int Put(int elno, int val) { if(elno == 1) el1.Put(val); else el2.Put(val); }
};
int main(void) {
	Collection coll;

	for(int i = 1; i <= 2; i++)
		coll.Put(i, i + 1);
	for(int i = 1; i <= 2; i++)
		cout << "Element #" << i << " = " << coll.Get(i) << endl;
	return 0;
}
```

Obiekty w obiektach

- Wniosek jest następujący: konstruktory obiektów wewnętrznych (obiektów przechowywanych w innych obiektach) są wywoływane, zanim konstruktory obiektów zewnętrznych rozpoczną swoją pracę.

```
#include <iostream>
using namespace std;
class Element {
      int value;
public:
      Element(void) { cout << "Element constructed!" << endl; }
      int Get(void) { return value; }
      void Put(int val) { value = val; }
};
class Collection {
      Element el1, el2;
public:
      Collection(void) { cout << "Collection constructed!" << endl; }
      int Get(int elno) { return elno == 1 ? el1.Get() : el2.Get(); }
      int Put(int elno, int val) { if(elno == 1) el1.Put(val); else el2.Put(val); }
};
int main(void) {
      Collection coll;
      return 0;
}
```

Obiekty w obiektach

- Konstruktor wywoływany niejawnie (czasami nazywany domyślnym konstruktorem) jest tym, który nie ma parametrów. Kompilator wyświetli komunikat o błędzie

```
#include <iostream>
using namespace std;
class Element {
      int value;
public:
      Element(int val) { value = val; cout << "Element(" << val << ") constructed!" << endl; }
      int Get(void) { return value; }
      void Put(int val) { value = val; }
};
class Collection {
      Element el1, el2;
public:
      Collection(void) { cout << "Collection constructed!" << endl; }
      int Get(int elno) { return elno == 1 ? el1.Get() : el2.Get(); }
      int Put(int elno, int val) { if(elno == 1) el1.Put(val); else el2.Put(val); }
};
int main(void) {
      Collection coll;
      return 0;
      }
```

Obiekty w obiektach

- Jeśli chcemy, aby konstruktor inny niż domyślny był wywoływany podczas tworzenia obiektu, który jest częścią innego obiektu możemy przedstawić go w następujący sposób:
  - Class(…) : inner_field_constr1(…), inner_field_constr2(…) { … }
- **Kiedy konstruktor jest dzielony między deklarację i definicję**, lista alternatywnych konstruktorów **powinna być powiązana z definicją**, a nie z deklaracją.

Obiekty w obiektach

- Poniższy kod jest poprawny:

```
class X {
    public:
        X(int z) { };
};
class Y {
        X x;
    public:
        Y(int z);
};
Y::Y(int z) : x(1) { };
```