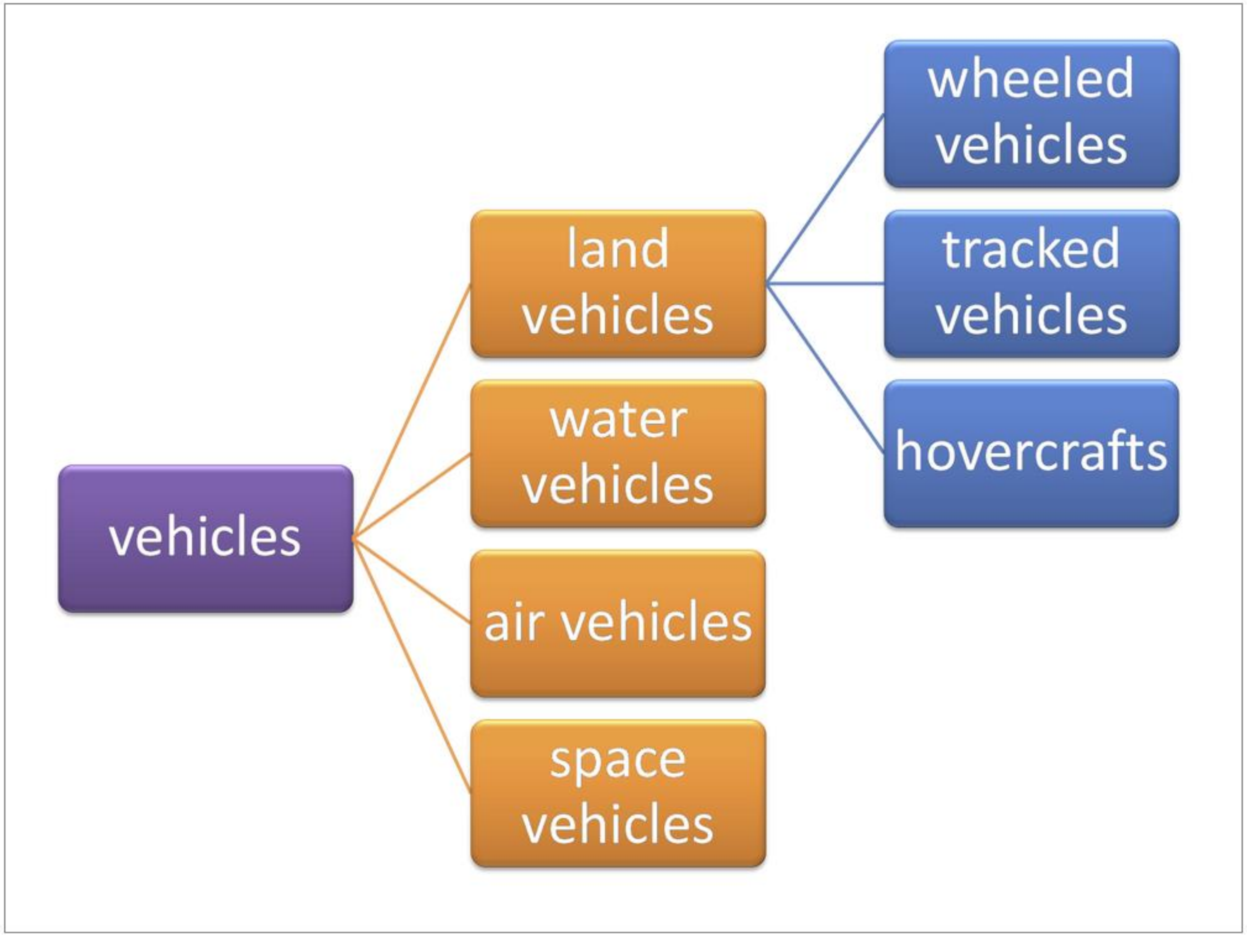
Maciej Sobieraj

# Lecture 4

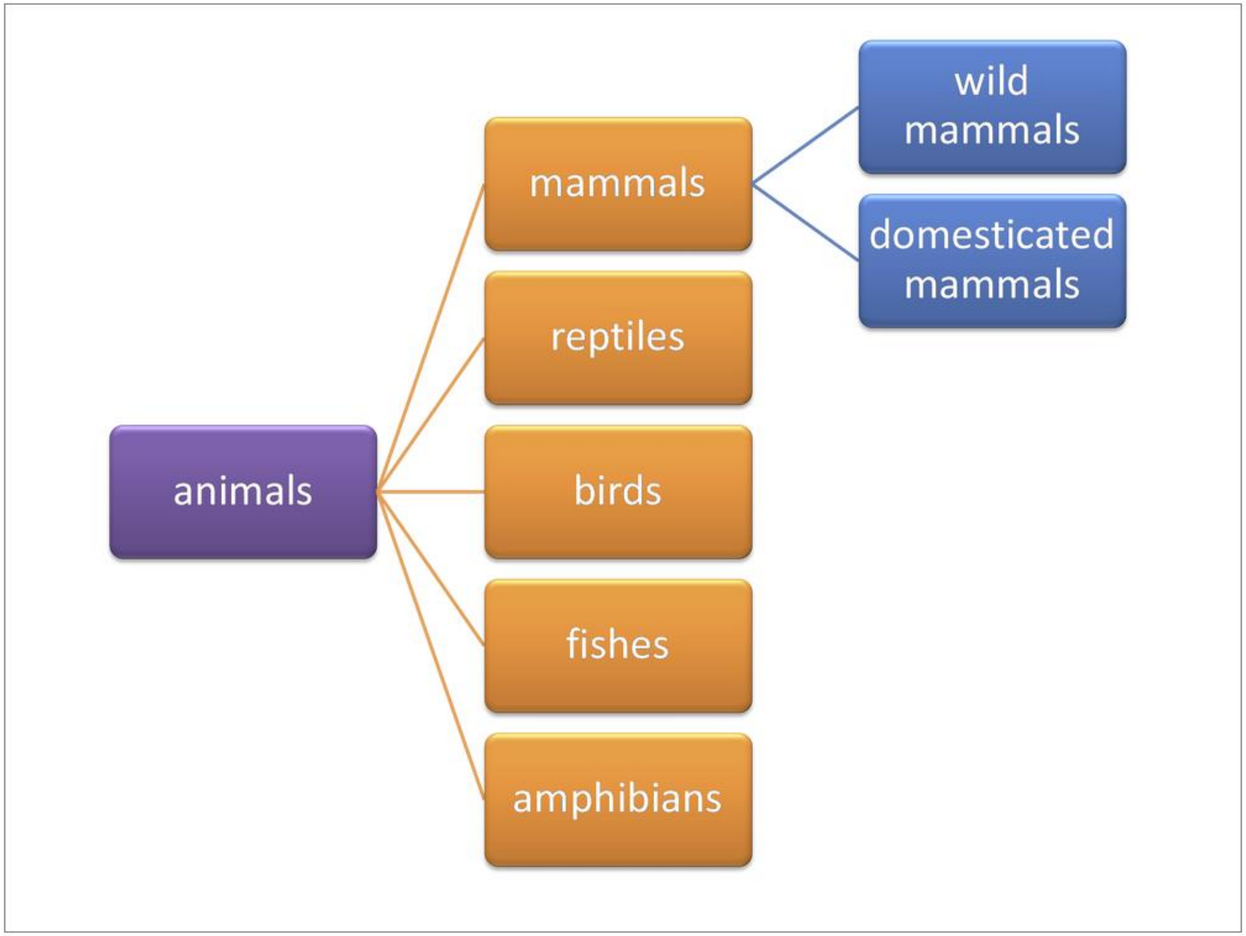
# Outline

1. **Object programming essentials**

# Class – what is it?



# Class – what is it?



# Object – what is it?

- A class is a set of objects.
- An object is a being belonging to a class.
- An object is an incarnation of requirements, traits and qualities assigned to a specific class.
- For example: any personal car is an object that belongs to the “wheeled vehicles” class.

# What does any object have?

- The object programming convention assumes that every existing object may be equipped with three groups of attributes:
  - an object has a name that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
  - an object has a set of individual properties that make it original, unique or outstanding (although there is the possibility that some objects may have no properties at all)
  - an object has a set of abilities to perform specific activities that can change the object itself or some of the other objects

# What does any object have?

- Here are two sample phrases that are good examples:
  - “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)

# Why all this?

- Object programming is the art of defining and expanding classes.
- Class is a model of a very specific part of reality reflecting properties and activities found in the real world.

```
class OurClass { };
```

# The very first object

- The newly defined class becomes an equivalent of a type, and we can use it as a type name.
- Imagine that we want to create one object of the *OurClass* class.

```
OurClass our_object;
```

# Stack aka LIFO

- A **stack** is a structure developed **to store data** in a very specific way.
  - The alternative name for a stack is **LIFO**. This is an abbreviation for a description of the stack’s behaviour: “**Last In – First Out**”.
- We also assume that the element at index 0 is at the bottom of the stack.

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

# Stack pointer

- We need a variable that can be responsible for storing **a number of elements currently stored** on the stack.

```
int SP = 0;
```

# Push

- We're now ready to define a function that **places a value** onto the stack.

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

# Pop

- Now it's time for the function to **take a value** off the stack.

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

# The stack in action

```
#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;
}
```

# Stack from scratch

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

# Stack from scratch

- This kind of data is called private in object programming. It’s private because only the class itself may access and modify it.
- If we want to mark some part of a class’s data as private, we have to add the keyword before the declarations.

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

# Stack from scratch

- Now it’s time for two functions that implement push and pop operations. The “C++” language assumes that a function of this kind (being a class activity) may be described in two different way:
  - **inside the class**, when the class body contains a full implementation of the method
  - **outside the class**, when the body contains only the function header; the function body is placed outside the class

# Stack from scratch

- We want to invoke functions to push and pop values. This means that they both should be accessible to every class's user This type of component is called "**public**" and we have to use a keyword to emphasize this fact.

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

# Stack from scratch

- The functions placed outside the class body need to be described in a very specific way. Their names should be **qualified** using the home class name and the “**::**” operator.

```
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;
}
```

# Stack from scratch

- Specialized function invoked implicitly every time a new stack is created.
  - **“constructor”** is **responsible for the proper construction of each new object of the class**.

```
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;
}
```

# Stack from scratch

- Note how we invoke a function from an object.
- This is the same convention we've already used for strings.

```
#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;
}
```

# Stack from scratch

- We want **a new stack with new capabilities**.
- In other words, we want to construct a **subclass** of the *Stack* class.

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

- Any object of the *AddingStack* class can do everything that each *Stack* class' object does.

# Stack from scratch

- We'll start with the implementation of the push function. This is what we expect from it:
  - to add the value to the sum variable
  - to push the value onto the stack

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

# Stack from scratch

- We’ll start with the implementation of the push function. This is what we expect from it:
  - to add the value to the sum variable
  - to push the value onto the stack

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

# Stack from scratch

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

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

# Stack from scratch

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

- Note the phrase “: Stack()”. It’s a request to invoke the **superclass constructor** before the current constructor starts its work.

# Stack from scratch

```
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;
}
```

# Stack from scratch

```
#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;
}
```

# Class components

- Since all the components are declared without the use of an **access specifier** (neither a *public* nor *private* keyword was added among the declarations) all three components are **private**. This means that a class defined in the following way:

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

- should be read as:

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

# Access specifiers

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

- The class has been rebuilt in order to show the use of **access specifiers**.

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

# Creating an object

Class the_object;

- The public components are available for use. You can do this:

  the_object.setVal(0);

- The private components are hidden and unavailable. You mustn't do this:

  ~~the_object.value = 0;~~

# Overriding component names

- The *setVal* function uses a parameter called *value*. The parameter overrides the class component called *value*.

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

# "this" pointer

- We assume that each object is equipped with a special component containing the following traits:
  - its name is **this**
  - it mustn’t be declared explicitly (it’s a keyword) so **it may not be overridden**
  - it’s **a pointer to the current object** – each object has its own copy of the *this* pointer

# "this" pointer

- The general rule says that:
  - if *S* is a **structure** or class and *S* has a **component** named *C* and
  - if *p* is a pointer to a structure of type *S*
  - then the *C* component may be accessed in the two following ways:
  - (*p).C  // *p* is explicitly dereferenced in order to access the *C* component
  - p->C    // *p* is implicitly dereferenced in order to access the *C* component

# "this" pointer

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

# Qualifying component names

- If any class function body is given outside the class body, its name must be qualified with the home class name and the "::" operator.

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

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

# Qualifying component names

- Class function names **may be overloaded** just like ordinary function names.
  - The first has one parameter and sets the *value* field with the value of the parameter.
  - The second has no parameters and sets the *value* field with -2.

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

# Constructors

- **A function with a name identical to its home class name is called a constructor**.
- The constructor is intended **to construct the object during its creation** i.e. to initialize field values, allocate memory, create other objects, etc.
- The constructor may access all object components like any other class member function but should not be invoked directly.

# Constructors

- The constructor **must not be declared using return type specifications**, including void type specifications.
- Declaring the object of the class, e.g. by doing it in the following way:
  - **Class** object;
- **implicitly invokes the constructor**.
- Note – you’re not allowed to do either this:
  - ~~object.Class()~~
  - ~~Class::Class();~~

# Constructors

```
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;

# Overloading constructor names

- Constructors may be overloaded too, depending on specific needs and requirements.

```
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;

# Overloading constructor names

- If a class has a constructor (or more precisely, at least one constructor), **one of them must be chosen** during object creation

```
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;~~

# Copying constructors

- There is a special kind of constructor intended to **copy one object into another**.
- If the copying constructor doesn't exist within a particular class and the initiator is actually used during the declaration of an object, its content will be actually (in a literal meaning) copied "field by field"
- Note that the keyword *const* used in the parameter declaration is **a promise** that the function won't attempt to modify the values stored in the referenced object.

# Copying constructors

```
#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;
}
```

# Memory leaks

- Failure to clean the memory will cause a phenomenon named "**memory leaking**", where the unused (but still allocated!) memory grows in size, affecting system performance.
- We can imagine that object creation consists of two phases:
  - the object itself is created and a part of the **memory is implicitly allocated** to the object
  - the constructor **explicitly allocates another part of the memory**
- Unfortunately, the memory explicitly allocated by the constructor **remains allocated**.

# Memory leaks

```
#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;
}
```

# Destructors

- We can safeguard ourselves against this danger by defining a special function called **destructor**. Destructors have the following restrictions:
  - if a class is named X, its destructor is named **~X**
  - a class can have **no more than one destructor**
  - a destructor **must be a parameter-less function** (note that the two last restrictions are the same – can you explain why?)
  - a destructor shouldn’t be invoked explicitly

# Destructors

```
#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;
}
```

# The “auto” keyword

- All the variables in your code belong to one of two categories. They are:
  - **automatic variables, created and destroyed**, sometimes repeatedly, and **automatically** (hence their name) during program execution
  - **static variables, existing continuously** during the whole program execution
- The “C” and “C++” programming languages assume that **all variables are automatic by default** unless they are declared explicitly as static.

# The “auto” keyword

```
#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;
}
```

# The “auto” keyword

- The *var* variable exists even when the *fun* function isn’t working, so the variable’s value is preserved between subsequent *fun* invocations.

```
#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;
}
```

# Instances of the class

- Every object created from a particular class is named a class's **instance**.
- None of these components really exist until the first instance is created.

```
#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;
	}
```

# Instances of the class

- All the rules on the previous slide are true if they refer to the **non-static components** of the class (both fields and functions).
- **A static component exists throughout the whole life of the program**. Moreover, there is always **only one component** regardless of the number of instances of the class.
- We can say that all the instances share the same static components.

# Static class variables

```
#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;
	}
```

# Static class variables

- This program produces the following output on the screen:
  - Static = 1, NonStatic = 10
  - Static = 2, NonStatic = 20

# Static class variables

- Note once again that the *Counter* field is accessed directly when it’s being used inside the class and with the “**::**” operator when it’s being used outside the class. It’s also possible to access the static variable through any of the existing class instances, like this:
  - cout << b.Counter ;

# Static class variables

```
#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;
	}
```

# Static class variables

- The program will output the following lines to the screen:
  - 2 instances so far
  - 4 instances
  - Bye, bye!

# Static class variables

```
#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;
	}
```

# Static class variables

- Note that any attempts to access the *Counter* variable expressed like this:
  - ~~Class::Counter = 1;~~
- are strictly **prohibited**.

# Static class variables

- The **static function**, like a static variable, may also be accessed (or more precisely, invoked) when no instances of the class have been created.
- Note that the static function may be invoked from inside the class, like this:
  - HowMany();
- or by using any of the existing instances, like this:
  - b.HowMany();

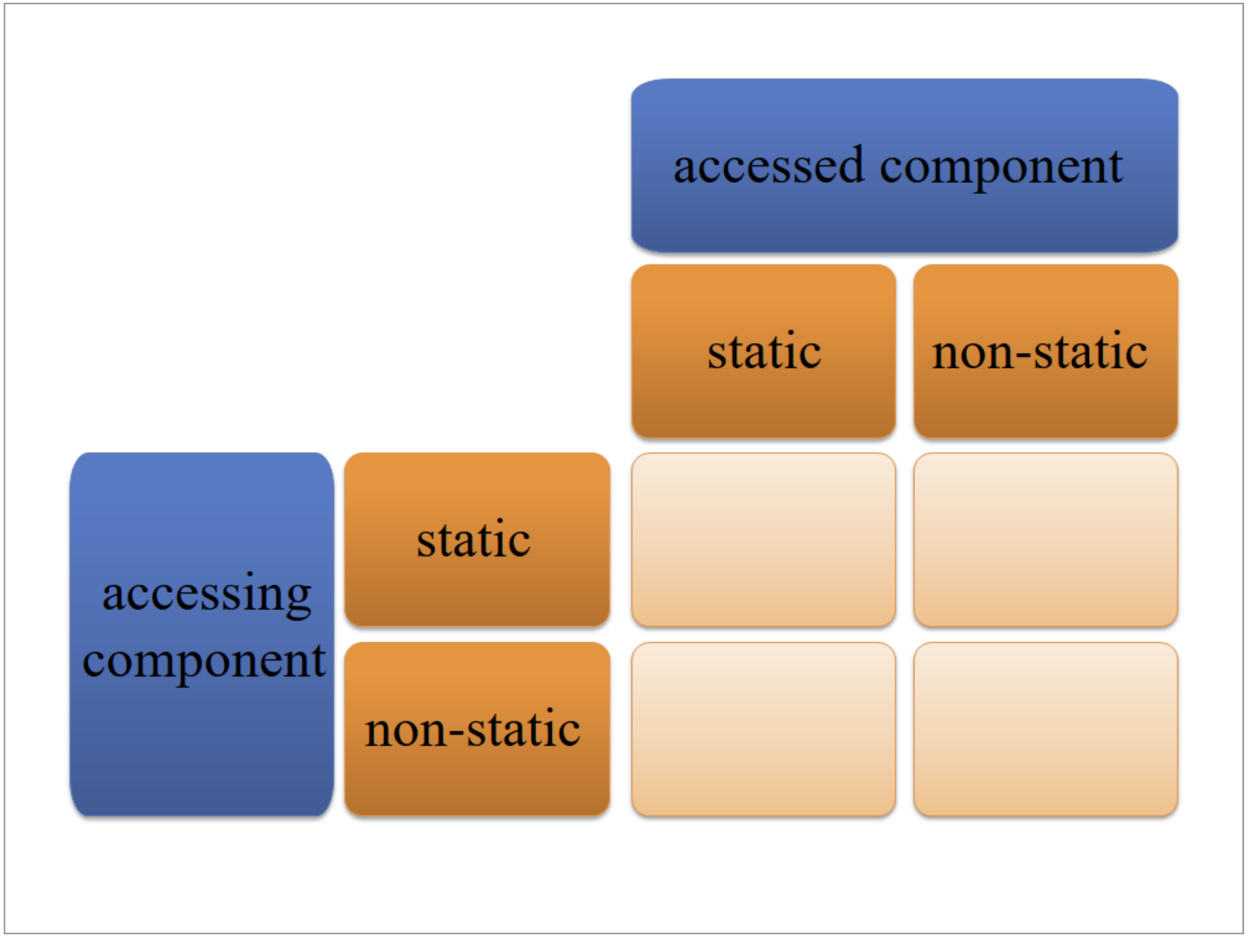
# Static class variables

```
#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;
	}
```

# Static vs. non-static components

- The coexistence of both static and non-static components within a single class causes some additional issues which we need to take into consideration. We can define four particular events when both types of components interact with one another.
- They are:
  - a **static** component accesses a **static** component
  - a **static** component accesses a **non-static** component
  - a **non-static** component accesses a **static** component
  - a **non-static** component accesses a **non-static** component

# Static vs. non-static components



# Static → static interaction

- The first test program (shown here →) demonstrates a case when **a static function named *funS2* tries to invoke another static function named *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;
}
```

# Static → static interaction

| | | | |
|---|---|---|---|
| | | accessed component | |
| | | static | non-static |
| accessing component | static | OK | |
| | non-static | | |

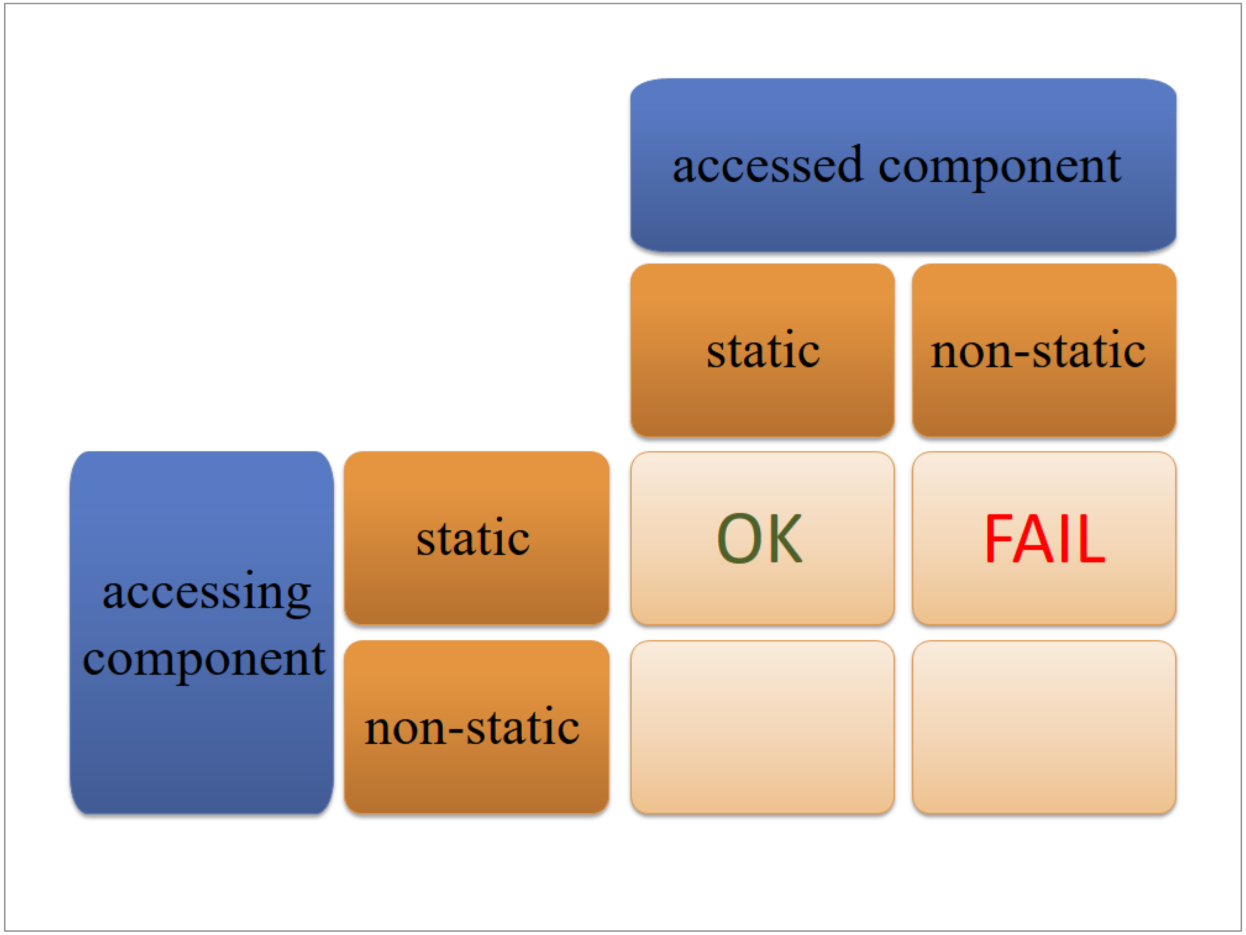
# Static → non-static interaction

- The second test program (shown here →) demonstrates a case when **a static function named *funS1* tries to invoke a non-static function named *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;
}
```

  - This program **cannot be successfully compiled**.

# Static → non-static interaction



# Nonstatic → static interaction

- The third test case (you can find it here →) refers to the situation where **a non-static function named *funN1* invokes a static function named *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();
}
```

# Nonstatic → static interaction

| | | accessed component | |
|---|---|---|---|
| | | static | non-static |
| accessing component | static | OK | FAIL |
| | non-static | OK | |

# Nonstatic → non-static interaction

- Is it possible to invoke a non-static function from within a non-static function

| | | accessed component | |
|---|---|---|---|
| | | static | non-static |
| accessing component | static | OK | FAIL |
| | non-static | OK | OK |

# Pointers to objects

- Objects may also exist as dynamically created and destroyed entities. In other words, **objects may appear on demand** – when they're needed – and vanish in the same way.

# Pointers to objects

```
#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;
	}
```

# Pointers to fields

- All the variables, including objects, brought to life in the “ordinary” way (by declaration, not by the use of the *new* keyword) live in **a separate area of memory called the *stack***. It’s a memory region dedicated to **storing all automatic entities**.
- The entities created “on demand” (by the *new* keyword) are created in **a specific memory region usually called a *heap***.

# Pointers to fields

- The object being stored in the heap must be accessed in a way that resembles the access to the dynamically allocated structures.
- You mustn’t use the ordinary “dotted” notation as there’s no structure (object) which can play the role of the left argument of the “.” operator unless you dereference the pointer.
- **You need to use the “arrow” (->) operator instead**.

# Pointers to fields

- The object being stored in the heap must be accessed in a way that resembles the access to the dynamically allocated structures.
- You mustn't use the ordinary “dotted” notation as there's no structure (object) which can play the role of the left argument of the “.” operator unless you dereference the pointer.
- **You need to use the “arrow” (->) operator instead**.

# Pointers to fields

```
#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;
}
```

# Pointers to functions

- Member functions invoked for an object accessed through the pointer have to be accessed using the arrow operator, too.

# Pointers to functions

```
#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;
}
```

# Selecting the constructor

- If a class has more than one constructor, one of them may be chosen during object creation. This is done by specifying the form of the parameter list associated with the class name.
- The list should be unambiguously compatible with one of the available class constructors.

# Selecting the constructor

```
#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;
}
```

# Arrays of pointers to objects

```
#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;
}
```

# Arrays of pointers to objects

```
#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;
}
```

# Objects inside objects

- **An object of any class may be the field of an object of any other class**.

```
#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;
}
```

# Objects inside objects

- The conclusion is: **constructors from inner objects (objects stored inside other objects) are invoked before the outer object's constructors start their work**.

```
#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;
}
```

# Objects inside objects

- The constructor invoked implicitly (sometimes called the **default constructor**) is the one which has no parameters. The compiler will produce an error message

```
#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;
	}
```

# Objects inside objects

- If we want **a constructor other than the default** one to be invoked during the creation of an object which is part of another object
- we can present it in the following schematic way:
  - Class(…) : inner_field_constr1(…), inner_field_constr2(…) { … }
- **When the constructor is divided between the declaration and the definition**, the list of alternative constructors should be **associated with the definition**, not the declaration.

# Objects inside objects

- The following snippet is correct:

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