C++ Software Engineering

for engineers of other disciplines

Module 2

"C++ OOP"

2nd Lecture: **`class OOP{};`**

ALTEN

Spring 2022

Gothenburg, Sweden

Object Oriented Programming

ALTEN

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Polymorphism

- Polymorphism in C++ :
  - Compile time a.k.a. static or early binding:
    - Templates
    - Function Overloading
  - Runtime a.k.a. Dynamic or late binding:
    - Base Class type deduction
    - Virtual and Pure Virtual Functions

polymorphism

/ˌpɒlɪˈmɔːfɪz(ə)m/

noun

noun: **polymorphism**; plural noun: **polymorphisms**

the condition of occurring in several different forms.

- OOP provides runtime polymorphism through inheritance.
- Although developers have better control over dynamic polymorphism, the statics are superior performance wise.
- Runtime is the time the program is executing (running).

" Polymorphism simply means the ability to use something without knowing everything about it. Rather than using a concrete definition, polymorphism relies on some form of prototype to define what types it takes. Any types that fit that prototype are accepted. "

https://stackoverflow.com/questions/10555864/what-is-the-difference-between-templates-and-polymorphism

Function Overloading – Early Binding

- Functions could be ***overloaded*** by taking different arguements as input.
- The ***compiler*** will pick the apropriate function based on the input prameter passed to it.

```
17   int main() {
18       Add obj;
19       obj.sum(10, 20);
20       sumOutsideClass(10,20);
21       obj.sum(11, 22, 33);
22       sumOutsideClass(11,22,33);
23       obj.sum(11,22,33,44);
24   }
```

```
class Add {
public:
  int sum(int num1, int num2) {
    return num1+num2;
  }
  int sum(int num1, int num2, int num3) {
    return num1+num2+num3;
  }
};
int sumOutsideClass(int num1, int num2) {
    return num1+num2;
}
int sumOutsideClass(int num1, int num2, int num3) {
    return num1+num2+num3;
}
```

```
test.cpp: In function 'int main()':
test.cpp:23:24: error: no matching function for call to 'Add::sum(int, int, int, int)'
   23 |     obj.sum(11,22,33,44);
      |                        ^
test.cpp:4:7: note: candidate: 'int Add::sum(int, int)'
    4 |   int sum(int num1, int num2) {
      |       ^~~
test.cpp:4:7: note:   candidate expects 2 arguments, 4 provided
test.cpp:7:7: note: candidate: 'int Add::sum(int, int, int)'
    7 |   int sum(int num1, int num2, int num3) {
      |       ^~~
test.cpp:7:7: note:   candidate expects 3 arguments, 4 provided
```

- If the compiler cannot find a proper function signature according to the input provided to the function, it will generate an error.
- Function overloading is not a feature exclusive to the `class` notation nor OOP paradigm.

Base Class Type Deduction

- ***Pointers*** to *derived* classes are type-compatible with a pointer of their *base* class.

- ***Objects*** of *derived* classes are type-compatible with their *base* class, yet assignement of an object of a derived class into a base class results in *object slicing*.

> *"In C++ programming, object slicing occurs when an object of a subclass type is copied to an object of superclass type: the superclass copy will not have any of the member variables defined in the subclass. These variables have, in effect, been sliced off".* https://en.wikipedia.org/wiki/Object_slicing

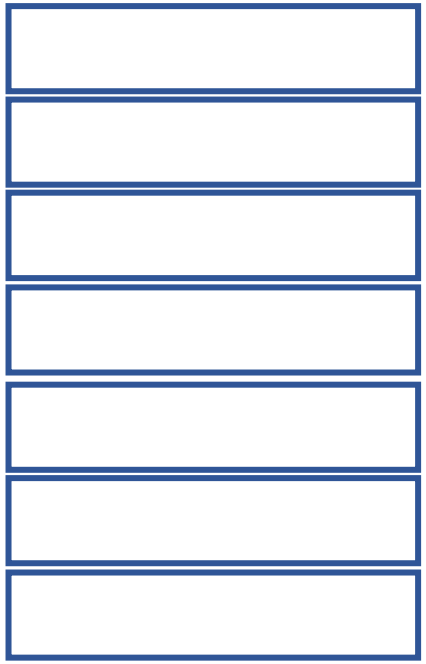
- There are very rare cases in which object slicing could be justified: https://stackoverflow.com/questions/2389125/object-slicing-is-it-advantage

```cpp
#include <iostream>
class BaseClass {
    // Class Body
};
class SomeChild : public BaseClass {
    // Class Body
};
class OtherChild : public BaseClass {
    // Class Body
};

void SomeFunction() {
    BaseClass  *base;
    SomeChild  first;
    OtherChild second;
    base = &first;
    base = &second;
    BaseClass *array[2];
    array[0] = &first;
    array[2] = &second;
}
```

Base Class Type


```
Polygon *p;
Triangle tria;
Rectangle rect;
p = &rect;
p->set_values(10,10);
p = &tria;
p->set_values(8,8);
```



Illustrations in this lecture are simplified and they do not resemble the actual reality. The details would be explained in later modules of this course.

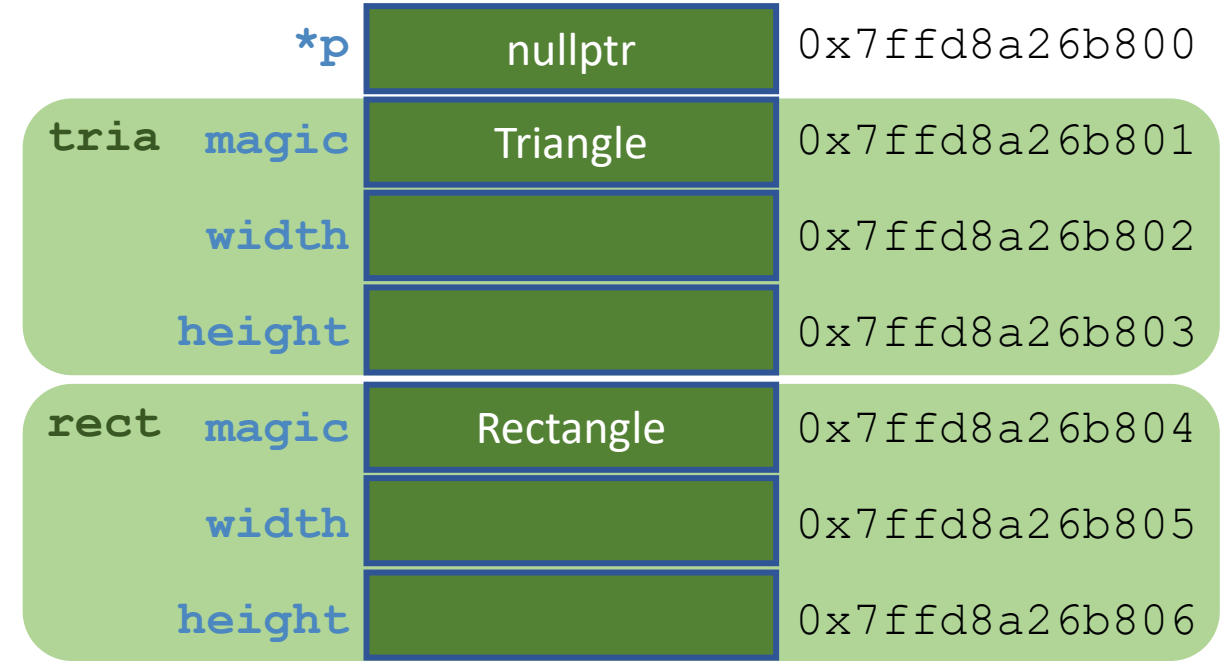
```
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Base Class Type


```
Polygon *p;
Triangle tria;
Rectangle rect;
p = &rect;
p->set_values(10,10);
p = &tria;
p->set_values(8,8);
```



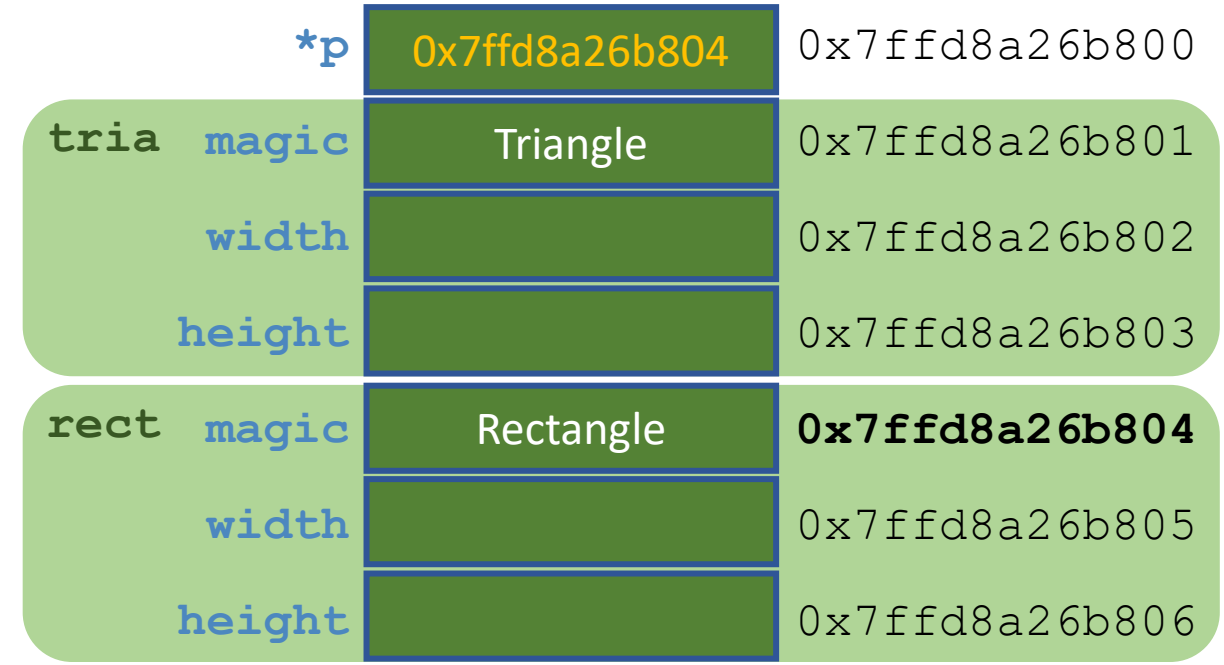

```
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Base Class Type


```
Polygon *p;
Triangle tria;
Rectangle rect;
p = &rect;
p->set_values(10,10);
p = &tria;
p->set_values(8,8);
```



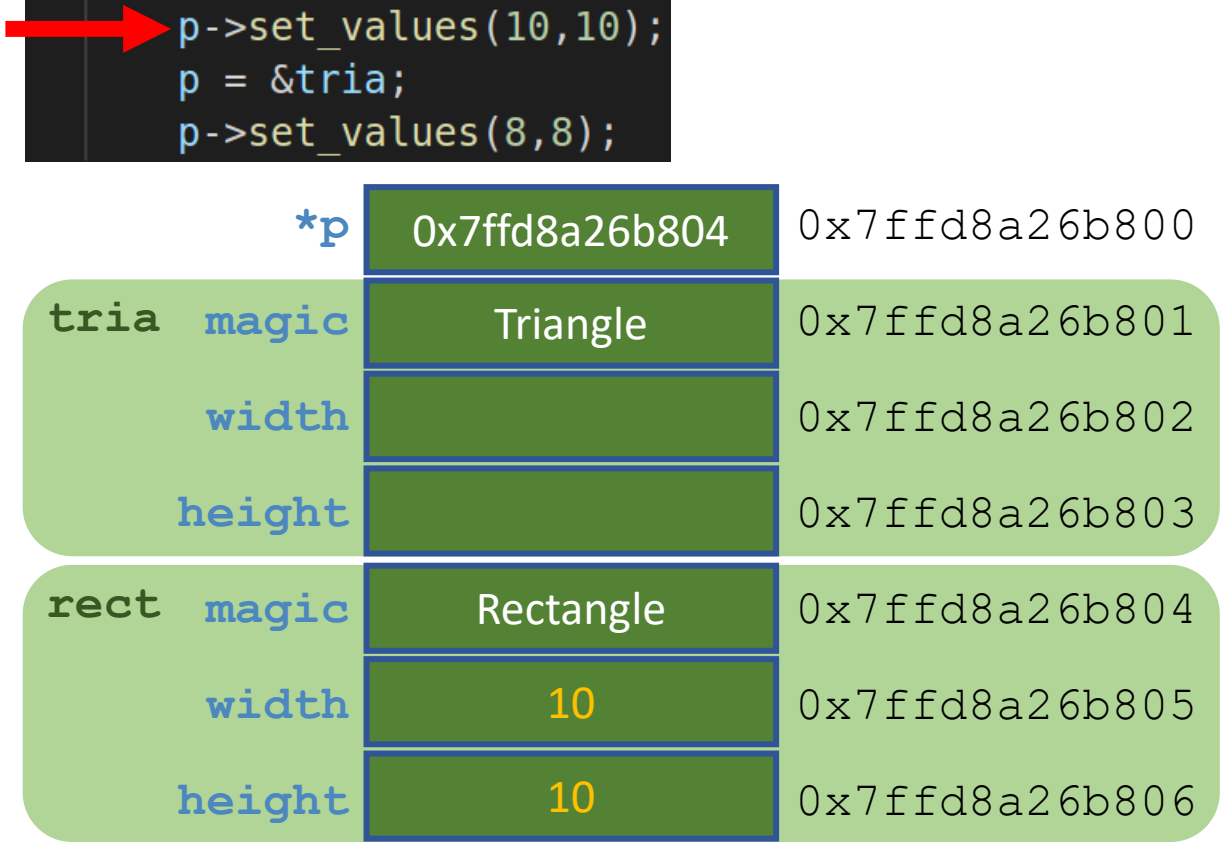

```
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Base Class Type


```
Polygon *p;
Triangle tria;
Rectangle rect;
p = &rect;
p->set_values(10,10);
p = &tria;
p->set_values(8,8);
```



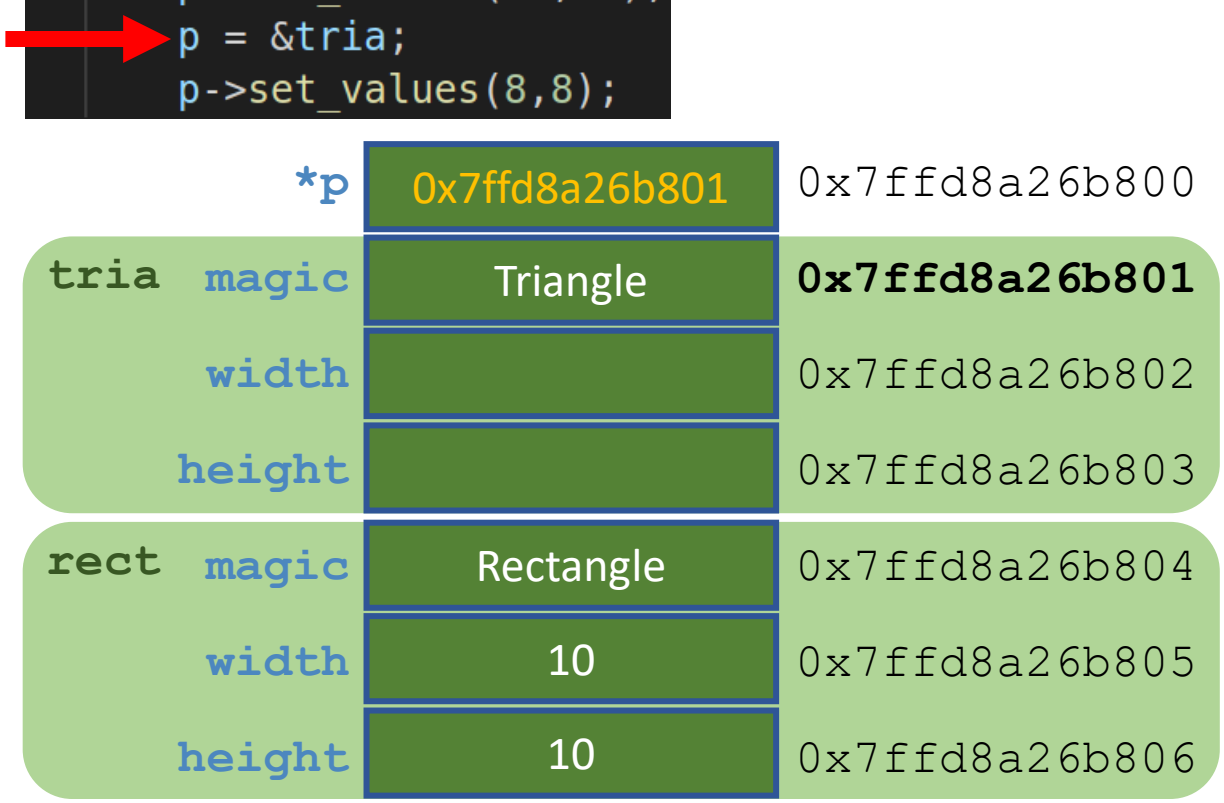

```
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Base Class Type


```
Polygon *p;
Triangle tria;
Rectangle rect;
p = &rect;
p->set_values(10,10);
p = &tria;
p->set_values(8,8);
```




```
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Base Class Type


```
Polygon *p;
Triangle tria;
Rectangle rect;
p = &rect;
p->set_values(10,10);
p = &tria;
p->set_values(8,8);
```

- Although both Rectangle and Triangle classes have the area function, it could not be called as the Polygon has no implementation of it!

	*p	0x7ffd8a26b801	0x7ffd8a26b800
tria	magic	Triangle	**0x7ffd8a26b801**
	width	8	0x7ffd8a26b802
	height	8	0x7ffd8a26b803
rect	magic	Rectangle	0x7ffd8a26b804
	width	10	0x7ffd8a26b805
	height	10	0x7ffd8a26b806

```
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Base Class Type

- In case both *base* and *derived* class implement the same function, the **datatype** of the object upon invocation of the function determines which implementation would be invoked.

```
int main() {
    Rectangle rect;
    Triangle tria;
    Polygon *p = new Polygon;
    std::cout <<p->area() << std::endl; // prints -1
    delete p;
    p = &rect;
    p->set_values(10,10);
    std::cout << p->area() << std::endl;// prints -1
    p = &tria;
    p->set_values(8,8);
    std::cout << p->area() << std::endl;// prints -1
}
```

- There is no way to reach the specific implementation of `area` function in *derived* classes from the *base* class pointer.

```
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
        { width=a; height=b;}
    int area() {return -1;}
};
class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Virtual Function

- Functions which ***could*** be *"redefined"* by derived classes.
- Virtual functions are declared as follows:

```
virtual ReturnDatatype FunName(…);
```

- During Runtime with the help of ***vtable*** the appropriate function would be invoked.

"Whenever a class defines a virtual function (or method), most compilers add a hidden member variable to the class that points to an array of pointers to (virtual) functions called the virtual method table. These pointers are used at runtime to invoke the appropriate function implementations, because ***at compile time it may not yet be known if the base function is to be called or a derived one*** *implemented by a class that inherits from the base class."*

https://en.wikipedia.org/wiki/Virtual_method_table

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
    virtual int area() { return -1; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Polymorphic Class

- A class defining or inheriting a `virtual` function is called a ***polymorphic*** class.
- Polymorphic classes could be instantiated.

```
int main() {
    Rectangle rect;
    Triangle tria;
    Polygon *p = new Polygon;
    std::cout <<p->area() << std::endl; // prints -1
    delete p;
    p = &rect;
    p->set_values(10,10);
    std::cout << p->area() << std::endl; // prints 100
    p = &tria;
    p->set_values(8,8);
    std::cout << p->area() << std::endl; // prints 32
}
```

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
    virtual int area() { return -1; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Pure Virtual Function

- Functions which ***must*** be *"redefined"* by the *concrete* derived classes.
- Pure virtual functions are declared as follows:

```
virtual ReturnDatatype FunName(...) = 0;
```

- Pure virtual functions fixate the input & output while present no actual implementation, hence refered to as *abstract method* since they abstract out the implementation.
- Pure virtual functions are mostly used in interfaces. An interface describes the behavior or capabilities of a C++ class without committing to an implementation of that class.

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
    virtual int area() = 0;
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Abstract Class

- A class defining a pure virtual function is called an ***abstract base*** class.
- Abstract classes could NOT be instantiated and must be used as *base* classes only.

```cpp
int main() {
    Rectangle rect;
    Triangle tria;
    Polygon *p = new Polygon;
    std::cout << p->area() << std::endl; // prints 32
}
```

```
test.cpp: In function ‘int main()’:
test.cpp:58:22: error: invalid new-expression of abstract class type ‘Polygon’
   58 |     Polygon *p = new Polygon;
      |                      ^~~~~~~
test.cpp:34:7: note:   because the following virtual functions are pure within ‘Polygon’:
   34 | class Polygon {
      |       ^~~~~~~
test.cpp:40:17: note:   ‘virtual int Polygon::area()’
   40 |     virtual int area() = 0;
      |                 ^~~~
```

```cpp
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
    virtual int area() = 0;
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Abstract Class


```
int main() {
    Polygon *p [2] = {
        new Rectangle(10,10),
        new Triangle(8,8) };
    for (size_t i = 0; i < 2; i++) {
        p[i]->print_area();
        delete p[i];
    }
}
```

- A call to a pure virtual or an abstract function within the abstract class is allowed. At runtime, the appropriate implementation of the function (depending on which object of derived class the pointer of the base class refer to) would be invoked.

```
class Polygon {
  protected:
    int width, height;
    virtual int area() = 0;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    void print_area() {
        std::cout << this->area() << std::endl;
    }
};
class Rectangle: public Polygon {
  public:
      Rectangle(int a,int b) : Polygon(a,b) {}
  private:
    int area () { return width * height; }
};
class Triangle: public Polygon {
  public:
      Triangle(int a,int b) : Polygon(a,b) {}
  private:
    int area () { return width * height / 2; }
};
```

Object Oriented Programming

ALTEN

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Message Passing

Object Oriented Programming

ALTEN

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Class-based vs Prototype-based

- Prototype-based programming uses generalized objects, which can then be cloned and extended.
- Class-based programming uses classes as definition for objects.

- Prototype-based programming allows for alteration of an object definition at runtime. Therefore, it is heavily dependent on runtime alteration, hence mostly uses late binding which decreases efficiency.
- C++ is a class-based OOP language.

"Using fruit as an example (in prototype-based programming), a "fruit" object would represent the properties and functionality of fruit in general. A "banana" object would be cloned from the "fruit" object and general properties specific to bananas would be appended. Each individual "banana" object would be cloned from the generic "banana" object." https://en.wikipedia.org/wiki/Prototype-based_programming

```
var foo = {one: 1, two: 2};

// bar.[[prototype]] = foo
var bar = Object.create(foo);

bar.three = 3;

bar.one; // 1
bar.two; // 2
bar.three; // 3
```

Object Oriented Programming

ALTEN

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Open Recursion

- Keyword **`this`** is used for open recursion.
- Open recursion is late-bound.

*"[**`this`**] allows a method defined in one class to invoke another method that is defined later, in some subclass thereof."* https://en.wikipedia.org/wiki/Object-oriented_programming#Open_recursion

- Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.
- Recursive invocation is when an object or a routine (function), either directly or indirectly, calls itself.

```
class Polygon {
  protected:
    int width, height;
    virtual int area() = 0;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    void print_area() {
        std::cout << this->area() << std::endl;
    }
};
class Rectangle: public Polygon {
  public:
      Rectangle(int a,int b) : Polygon(a,b) {}
  private:
    int area () { return width * height; }
};
class Triangle: public Polygon {
  public:
      Triangle(int a,int b) : Polygon(a,b) {}
  private:
    int area () { return width * height / 2; }
};
```

Object Oriented Programming

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Dynamic Dispatch & Message Passing

ALTEN

- Distinguishing an object from an abstract data type at runtime
- Invoking methods of an object is conceptualized as a message (with some input parameters at times) sent to the object, thus function call is a.k.a. Message passing.

```
int main() {
    Polygon *p [2] = {
        new Rectangle(10,10),
        new Triangle(8,8) };
    for (size_t i = 0; i < 2; i++) {
        p[i]->print_area();
        delete p[i];
    }
}
```

```
class Polygon {
  protected:
    int width, height;
    virtual int area() = 0;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    void print_area() {
        std::cout << this->area() << std::endl;
    }
};
class Rectangle: public Polygon {
  public:
        Rectangle(int a,int b) : Polygon(a,b) {}
  private:
    int area () { return width * height; }
};
class Triangle: public Polygon {
  public:
        Triangle(int a,int b) : Polygon(a,b) {}
  private:
    int area () { return width * height / 2; }
};
```

`struct` vs `class`

- To define a class (object) either `struct` or `class` keyword could be used.
- Default access for members of `class` is `private`, as it is for inheritance.
- Default access for memebrs of `struct` is `public`, as it is for inheritance.

```
test.cpp:38:7: error: redefinition of ‘class foo’
   38 | class foo {
      |       ^~~
test.cpp:35:8: note: previous definition of ‘class foo’
   35 | struct foo {
      |        ^~~
```

```
35  struct foo {
36      foo(){}
37  };
38  class bar {
39      bar(){}
40  };
41  void SomeFunction() {
42      foo f;
43      bar b;
44  }
```

- Although both `class` and `struct` keywords could be used interchangeably, yet the keyword `struct` is generally used to specify plain data structures.
- Classes could also be defined using `union`, however only one data member is accessible at a time.

```
35  struct foo {
36      foo(){}
37  };
38  class foo {
39      foo(){}
40  };
```

```
test.cpp: In function ‘void SomeFunction()’:
test.cpp:43:9: error: ‘bar::bar()’ is private within this context
   43 |     bar b;
      |         ^
test.cpp:39:5: note: declared private here
   39 |     bar(){}
      |     ^~~
```

enum class

- To create ***real*** enumeration type in C++ keyword **`enum class`** is used.
- Unlike **`enum`**s they are not *implicitly* convertible to **`int`** .

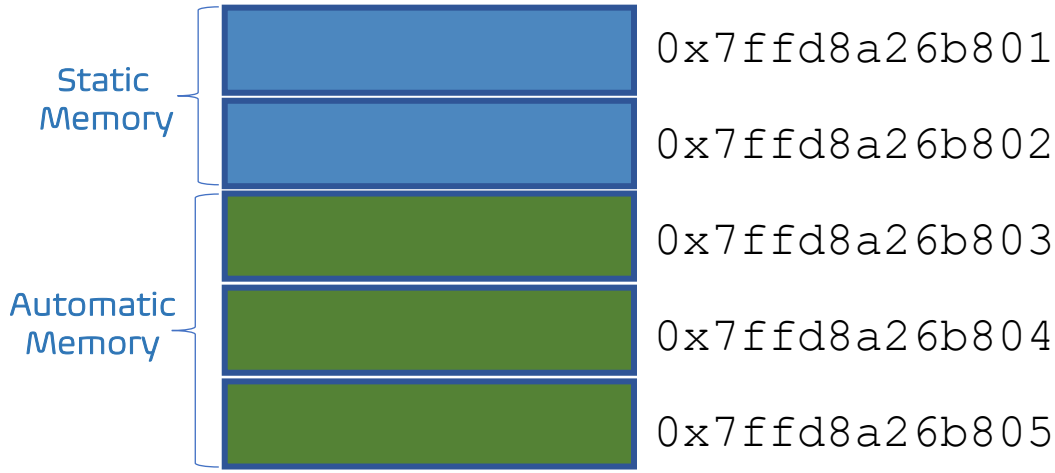
```
enum class WeekDays {Saturday, Sunday, Monday, Tuesday,
                     Wedensday, Thursday, Friday};
enum class Colors : char {black, blue, green, red,
                          purple, yellow, white, null};
Colors WhatColorToWear(WeekDays _day) {
    Colors appreanceColor = Colors::null;
    if (_day == Colors::black) {
        appreanceColor = Colors::black;
    } // ....
    switch (_day) {
    case WeekDays::Saturday:
        appreanceColor = Colors::black;
        break;
        // ...
    default:
        break;
    }
    return appreanceColor;
}
```

- The actual type used for the enumeration type could be specified using colon – please refer to the **`Colors`** enumeration in the picture for an example; each color enumeration is a **`char`**.

static

- Variable declared with keyword **`static`** are stored in *static* memory while the program is executing.

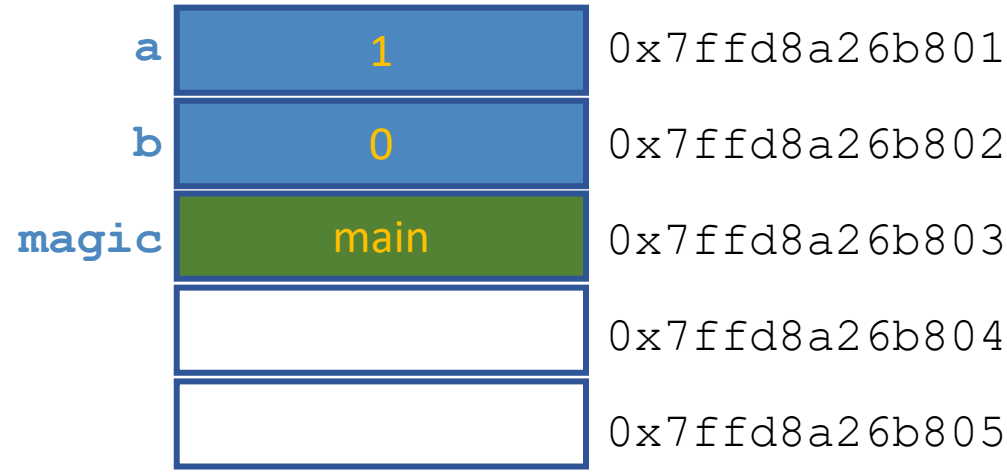
```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```



- Static memory is available throughout the execution of the program. Static variables' value stays in the static memory, even when they are out of scope.
- Variable stored in automatic memory, however, will be removed from memory once they go out of scope.
- Static variables could only be initialized using constant values.

static


```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```



- Static variables if not initialized, are set to zero (i.e. default constructor is invoked).

Illustrations in this lecture are simplified and they do not resemble the actual reality. The details would be explained in later modules of this course.

static


```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

- When a function is loaded into *call stack*, its local variables would be loaded into the frame, most compilers would initialize them with the variable from the beginning.

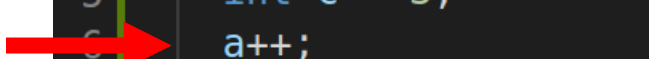
static

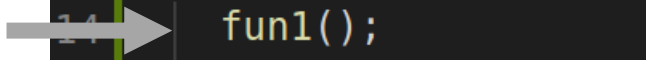

```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

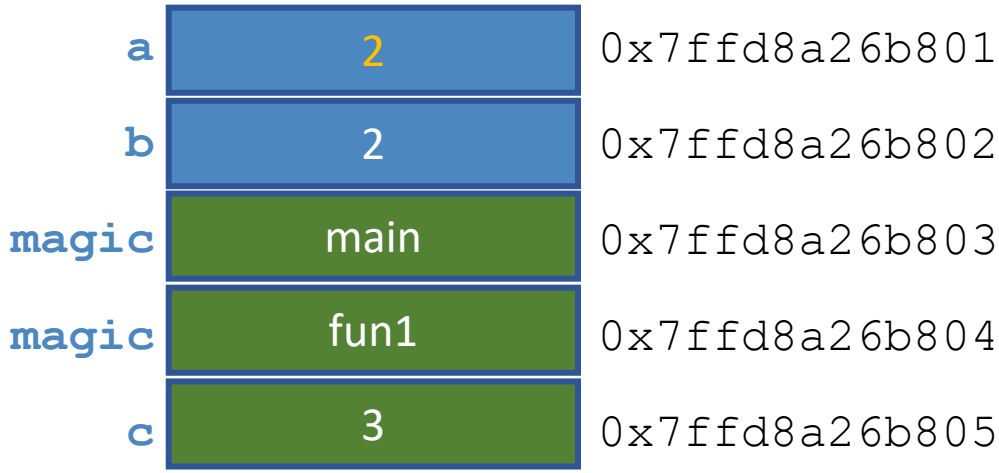
- Static variables inside a function, would be initialized first time the execution reaches to them.

static


```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

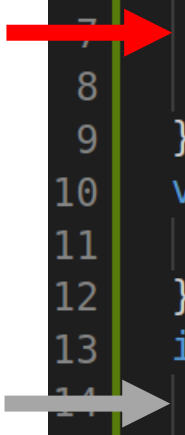


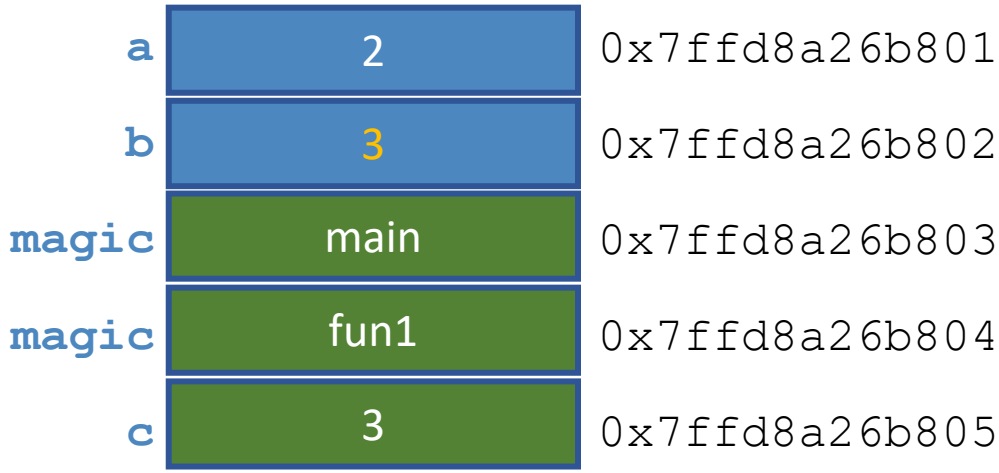




static

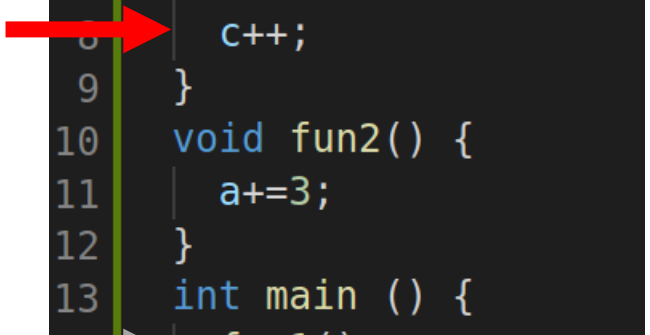

```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

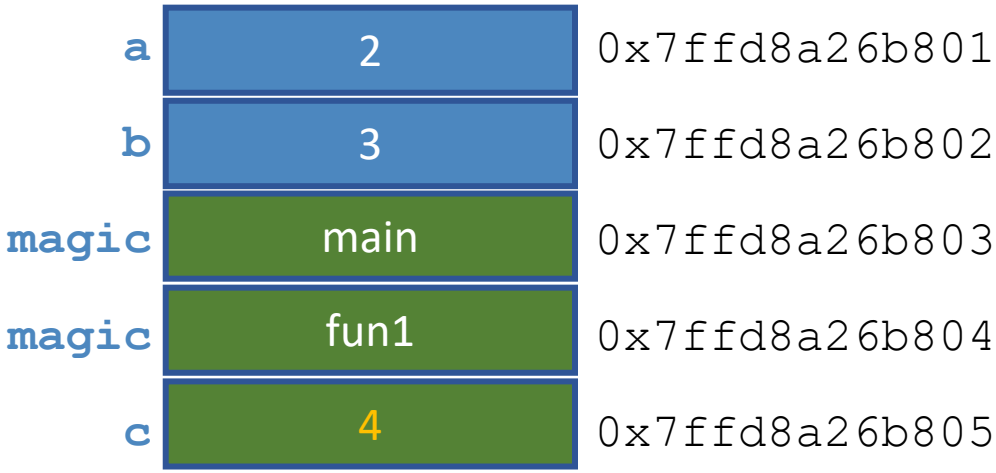




static

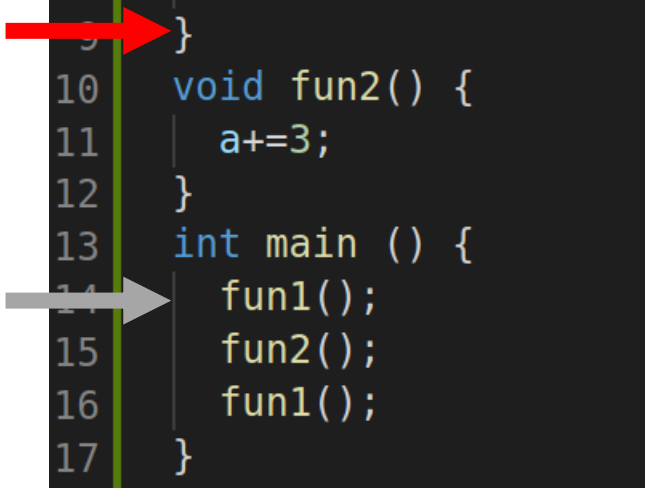

```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

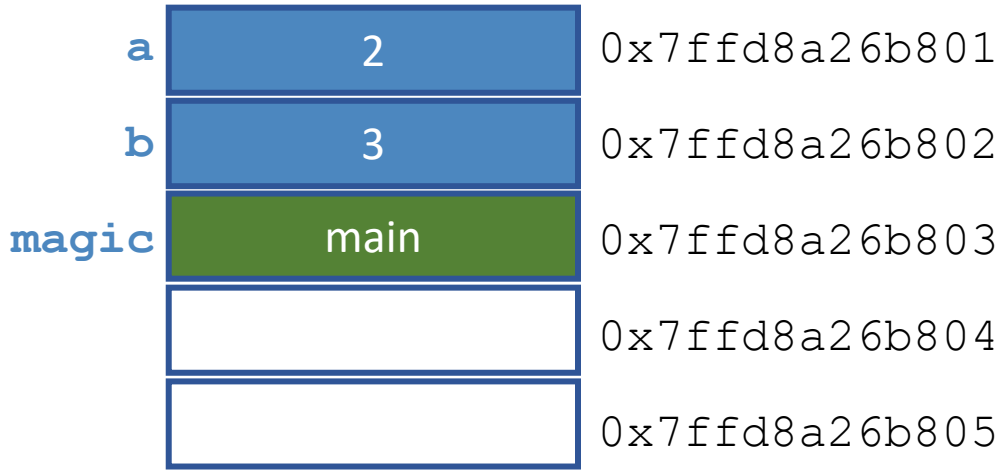




static


```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

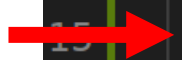


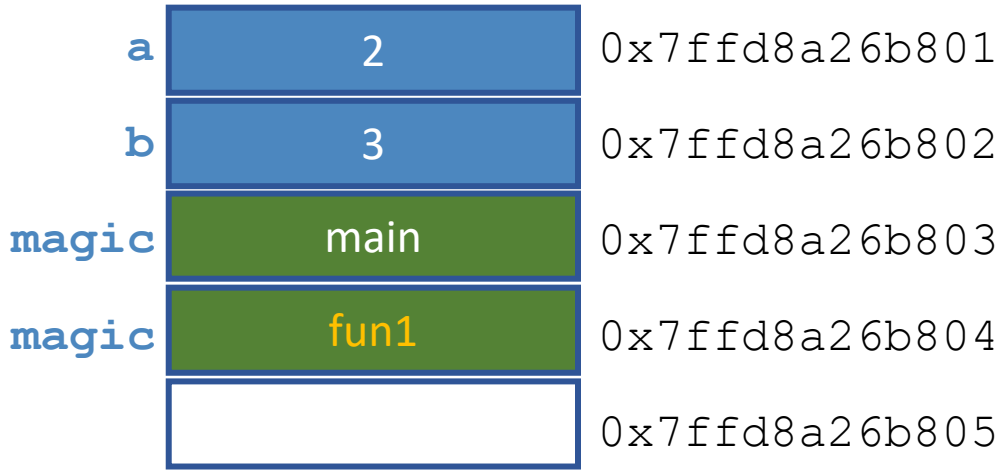


- Local variables allocated in *automatic memory* will be automatically removed once the function frame is removed from *call stack*.

static


```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```





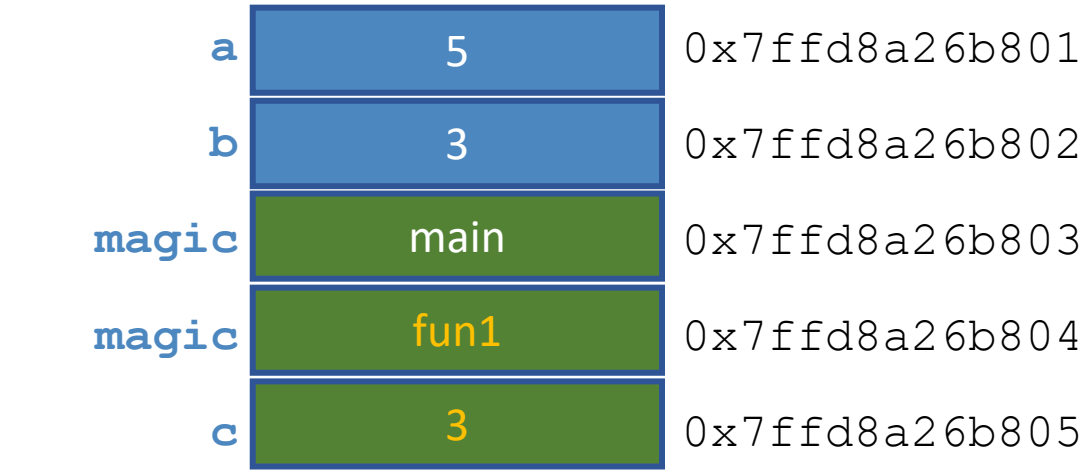
static


```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

- Out of scope **`static`** variables could not be reached yet they *preserve* the value.

static


```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

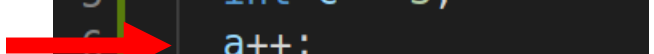


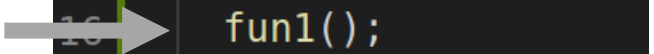
- Static variables insides a method are only initialized the first time execution reaches to them, the following calls would skip the initialization.

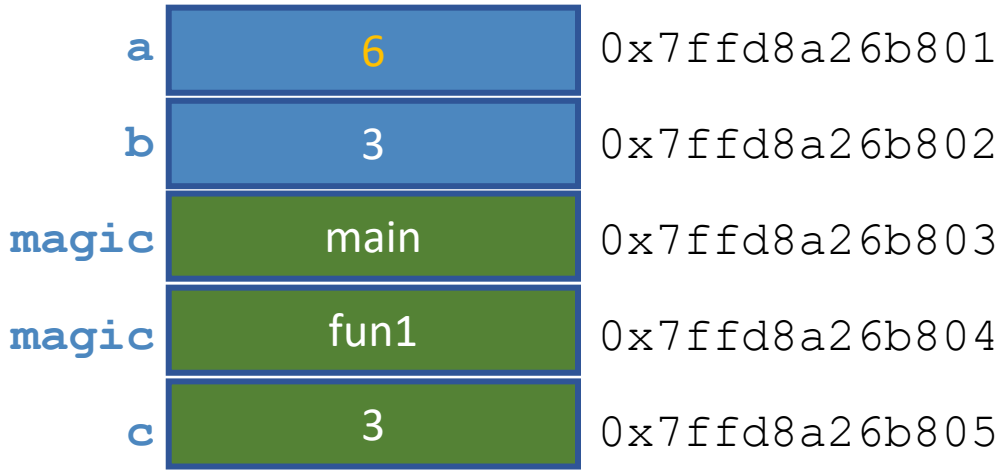
static

ALTEN

```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

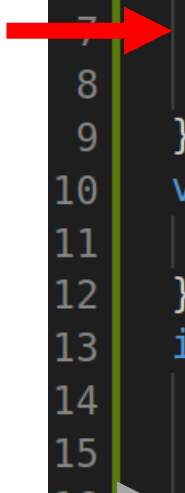


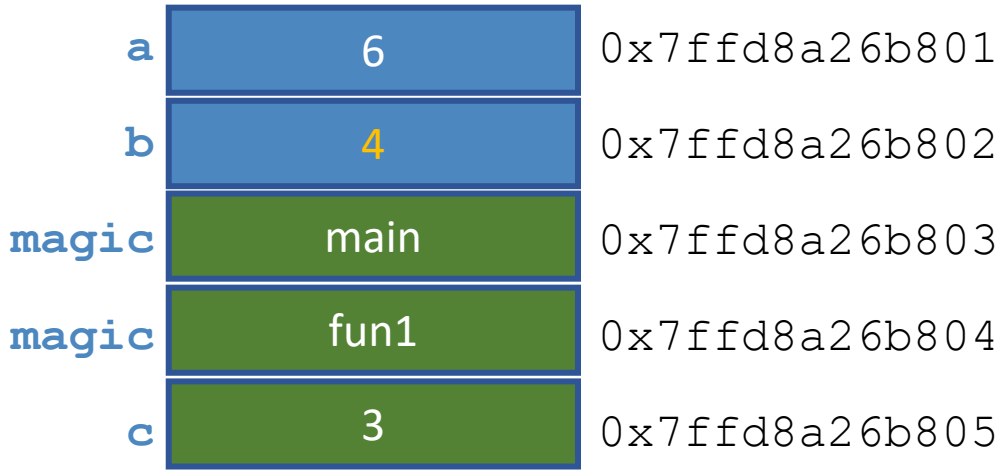




static

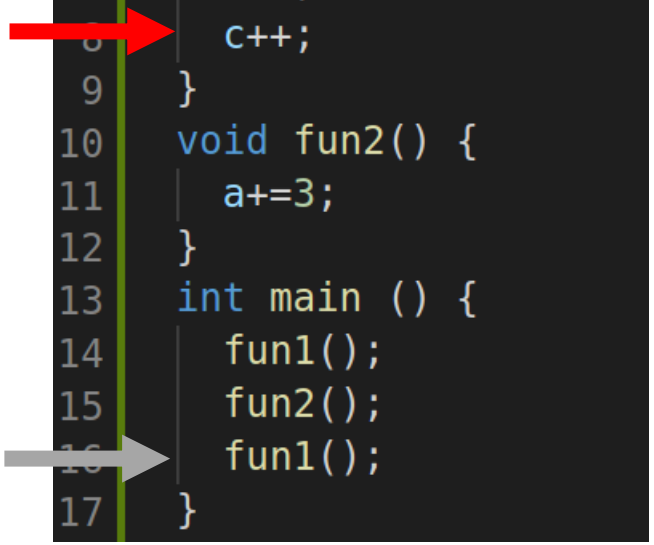

```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

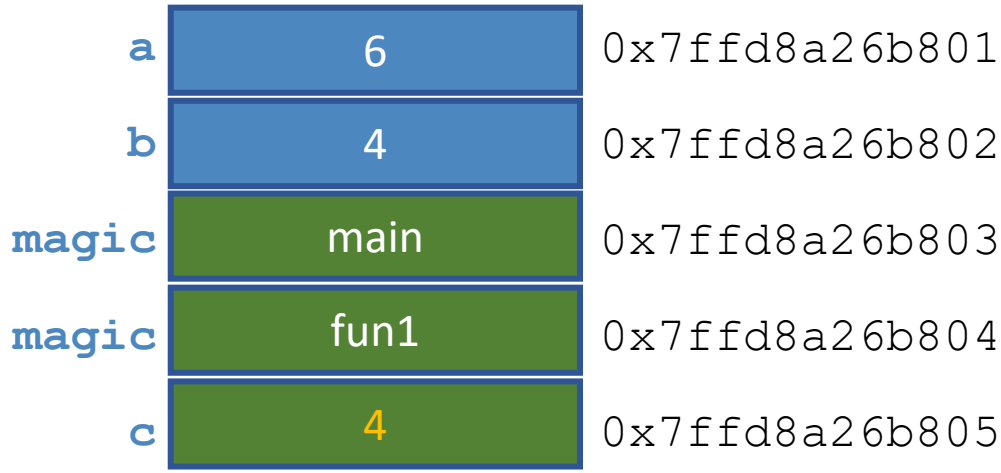




static

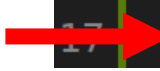

```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```

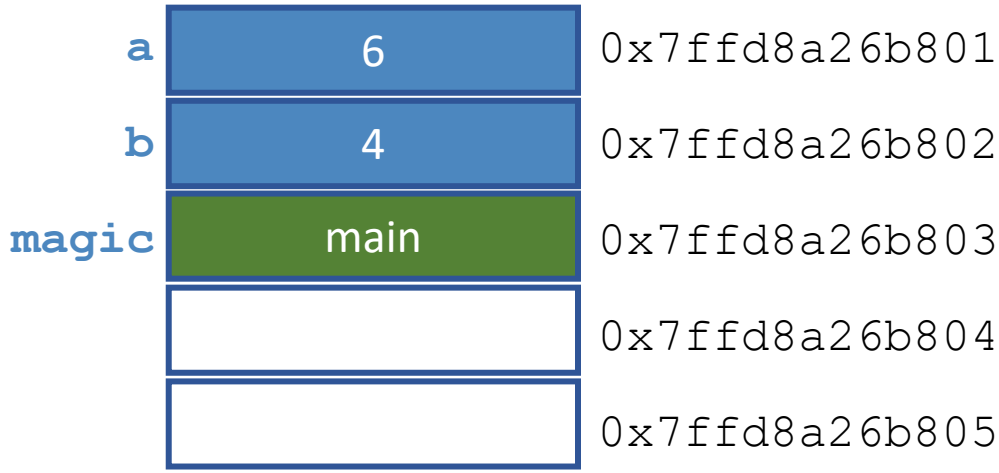




static


```
1   #include <iostream>
2   static int a = 1;
3   void fun1() {
4     static int b = 2;
5     int c = 3;
6     a++;
7     b++;
8     c++;
9   }
10  void fun2() {
11    a+=3;
12  }
13  int main () {
14    fun1();
15    fun2();
16    fun1();
17  }
```





static

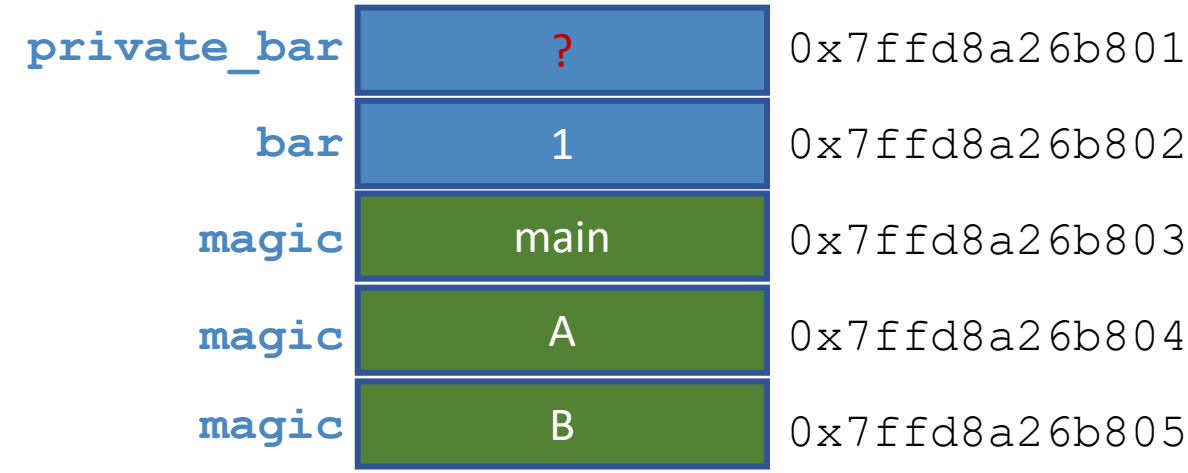
- Classes can also have members declared with keyword **`static`** to store them in *static* memory.

```
#include <iostream>

class Foo {
  static int private_bar;
public:
  static int bar; // Declaration
  void setPrivateBar() {
    this->private_bar = 1;
  }
};

int Foo::bar = 1; // Definition

int main () {
  Foo A,B;
  return 0;
}
```



- Static variables insides a class are not associated with the object of the class as they are allocated in different region of memory and treated as independent variables.
- Static variables are *shared* by all the objects of the class, and cannot be initialized by the constructor, and shall be initialized explicitly.
- Static variables must be defined separately (in source code), and do not need to have **`static`** keyword any more.

DEMO!

Base Class Type Deduction

ALTEN

- It is only the pointer of the base class which could be instantiated or used as a type.

```
class Dog : public Animal {
public:
    Dog() = default;
    virtual void speak() {
        std::cout << "Woof" << std::endl;
    }
};

class Chiwawa : public Dog {
public:
    Chiwawa() = default;
    void speak() {
        std::cout << "Soft_Woof" << std::endl;
    }
};
```

```
Animal **p;

p = new Animal*[3];
p[0] = new Cat;
p[1] = new Dog;
p[2] = new Chiwawa;

for (size_t i = 0; i < 3; i++) {
    p[i]->speak();
```

```
class Animal {
public:
    virtual void speak() = 0;
};

class Cat : public Animal {
public:
    Cat() = default;
    void speak() {
        std::cout << "Meow" << std::endl;
    }
};
```

final & override


```cpp
class Base {
public:
    virtual void vMethod() {
        std::cout << "This is vMethod from Base Class!" << std::endl;
    }
    virtual void pMethod() = 0;
};
class Foo : public Base {
public:
    void vMethod() override /*optional*/ {
        std::cout << "This is vMethod from Foo!" << std::endl;
    };
    void pMethod() final {
        std::cout << "This is pMethod from Foo!" << std::endl;
    }
    void nonVirt() {
        std::cout << "This is nonVirt from Foo!" << std::endl;
    }
};
```

```cpp
class Bar final : public Foo {
public:
    int vMethod() override { // Different signature from Foo::vMethod()
        std::cout << "This is vMethod from Bar!" << std::endl;
    };
    void pMethod() { // Final function
        std::cout << "This is pMethod from Foo!" << std::endl;
    }
    void nonVirt() override { // Non virtual function
        std::cout << "This is nonVirt from Foo!" << std::endl;
    }
};

class err : public Bar /*Bar is final*/{};
```

virtual

ALTEN

```cpp
class Foo {
    /*virtual*/ void private_fun() {
        std::cout << "Foo Private" << std::endl;
    }
 public:
    void interface_fun() {
        std::cout << "Foo Interface" << std::endl;
        private_fun();
    }
};

class Bar : public Foo {
public:
    void private_fun(){ //overrides
        std::cout << "Bar override private"<< std::endl;
    }
    void interface_fun(int) { //hides
        std::cout << "Bar override interface"<< std::endl;
    }
};
```

```cpp
class B : public Bar {
public:
    void interface_fun() {
        std::cout << "B override interface" << std::endl;
        Foo::interface_fun();
    }
};

class BB : public Bar {};
class FF : public Foo {};
```

- Methods need to be **`virtual`** for dynamic dispatch.

```cpp
struct A {
    virtual void f();
};
struct VB1 : virtual A {
    void f(); // overrides A::f
};
struct VB2 : virtual A {
    void f(); // overrides A::f
};
// struct Error : VB1, VB2 {
//     // Error: A::f has two final overriders in Error
// };
struct Okay : VB1, VB2 {
    void f(); // OK: this is the final overrider for A::f
};
struct VB1a : virtual A {}; // does not declare an overrider
struct Da : VB1a, VB2 {
    // in Da, the final overrider of A::f is VB2::f
};
```

https://en.cppreference.com/w/cpp/language/virtual

Abstract


```cpp
class Abstract {
public:
    virtual void f() = 0;
    virtual void g() {
        std::cout << "Abstract::g" << std::endl;
    };

    ~Abstract() {
        g();
        //f(); UDB
        Abstract::f();
    }
};
void Abstract::f() {
    std::cout << "Abstract::f" << std::endl;
}
```

https://en.cppreference.com/w/cpp/language/abstract_class

```cpp
class Concrete : public Abstract {
public:
    virtual void f() {
        std::cout << "Concrete::f()" << std::endl;
    }
    virtual void g() {
        std::cout << "Concrete::g()" << std::endl;
    }
    ~Concrete() {
        g();
        f();
    }
};
class Abstract2 : public Concrete {
public:
    virtual void g() = 0;
};
```

Virtual Destructor


```cpp
int main() {
    A *_ = new B;
    delete _;
    std::cout << " ----- " << std::endl;
    B b;
    return 0;
}
```

```cpp
class A {
public:
    //virtual void f() = 0;
    /*virtual*/ ~A() {
        std::cout << "doing nothing!" << std::endl;
    }
};
class B : public A {
    char *p = new char();
    //void f(){}
public:
    ~B(){
            std::cout << "cleaning up!" << std::endl;
            delete p;
        }
};
```

enum class


```
enum Mode {INT, DBL};
enum Type {A, B};

enum class MODE {INT, DBL};
enum class TYPE {A,B};
```

```
/*
    MODE m = MODE::INT;
    TYPE t = TYPE::A;
*/
    Mode m = Mode::DBL;
    Type t = Type::B;
    printMode(m);
    if (m == t) { }
    switch (t) {
    case Mode::DBL:
        printMode(Mode::DBL);
        break;
    case Mode::INT:
        printMode(Mode::INT);
        break;
    default:
        std::cout << "Nothing" << std::endl;
        break;
    }
```

```
void printMode (Mode _m ){
    std::cout << "Mode is: " << _m << std::endl;
}

void printMode (MODE _m ){
    std::cout << "Mode is: " <<
    static_cast<std::underlying_type<MODE>::type>(_m) << std::endl;
}
```

static


```
class Foo {
public:
    static int static_int /*= 0*/;
    int        normal_int = 9;
    static void printStatic() {
        std::cout << /*this->*/static_int << std::endl;
//      std::cout << /*this->*/ normal_int << std::endl;
    }
    void printNormal() {
        std::cout << this->static_int << std::endl;
        std::cout << this->normal_int << std::endl;

    }
};
```

```
int Foo::static_int = 0;

int main () {
    Foo::printStatic();
//  Foo::printNormal();
    std::cout << " -------- " << std::endl;
    Foo A,B;
    A.static_int ++;
    A.normal_int --;

    A.printNormal();
    std::cout << " -------- " << std::endl;
    B.printNormal();

    return 0;
}
```

Assignment 1

- Implement an *interface* for shape, and derive circle, triangle, rectangle, square as *children.* Overload all three comparison operators to compare the shapes based on their area -- if equal then their perimeter should be considered. Only, and if only both area and perimeter are equal then the objects should be considered equal.

- Take Pi=3.

Bonus

- Identify the abstract objects from yesterday's assignment and implement *interfaces* for them – update the code with today's lecture topics.