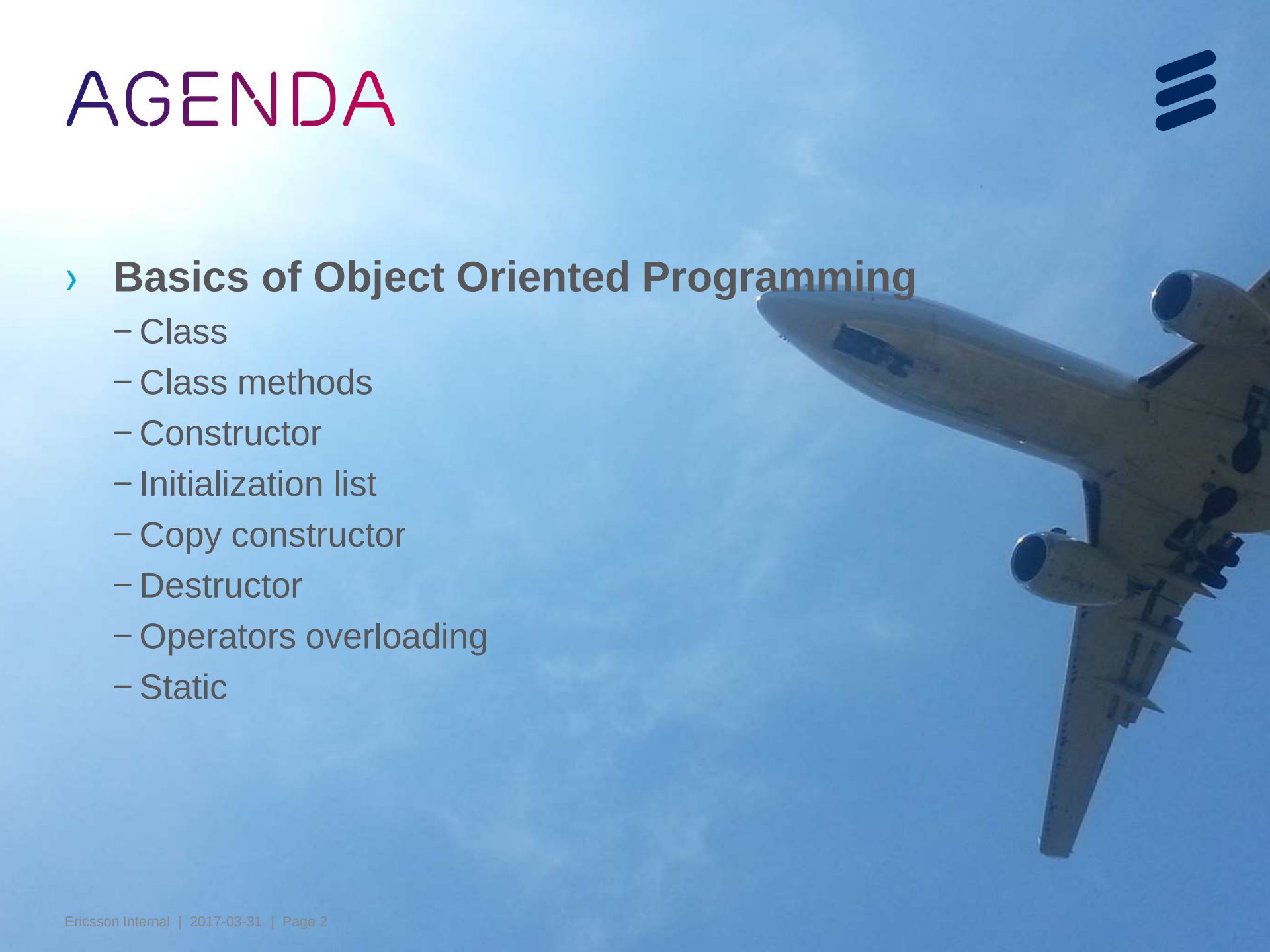
INTERMEDIATE C++

Trener: Michał Rad

AGENDA



- **Basics of Object Oriented Programming**
  - Class
  - Class methods
  - Constructor
  - Initialization list
  - Copy constructor
  - Destructor
  - Operators overloading
  - Static

AGENDA

- **Inheritance**
  - Derived classes
  - Derived classes constructors
  - Base class members in derived class
  - Types of inheritance
- **Polymorphism**
  - Virtual methods
  - Abstract classes

AGENDA

- **Type Casting**
  - C++ advanced casting mechanisms
- **Exceptions**
  - Exceptions handling
  - Creating exceptions
- **Templates**
  - Function templates
  - Introduction to STL

CHAPTER I
BASICS OF OBJECT ORIENTED PROGRAMMING

BASICS OF OBJECT ORIENTED PROGRAMMING

Towards Abstraction!

?

Declarative Languages (Prolog, SQL...)

OO Languages (C++, Java...)

Procedural Languages (C, Fortran...)

Assembly Language

Binary Code

BASICS OF OBJECT ORIENTED PROGRAMMING

Objects model elements of the problem.

Each object has a set of characteristics and responsibilities (functionality / behaviour).

Problem: Design a soccer manager game.

SOCCER PLAYER
name, age, height, weight, position, salary, speed skill, shooting skill, passing skill, ball control skill, statistics, etc.
shoot on goal, pass the ball, jump into air, wave to the fans, punch another player, etc.

Characteristics

(variables)

Responsibilities

(functions)

BASICS OF OBJECT ORIENTED PROGRAMMING

Class define new type of data structure and the functions that operate on those data structures.

Object is an instance of the class.

BASICS OF OBJECT ORIENTED PROGRAMMING

Class Declaration and Definition works very similar to declaration and definition of a single function.

Class Declaration (often placed in header file) consist of list of functions (methods) and members (variables) the class provides.

Class Definition (often placed in source file) implements the functions (methods) of the class.

BASICS OF OBJECT ORIENTED PROGRAMMING

Class Declaration (Rectangle.h)

class keyword

class name

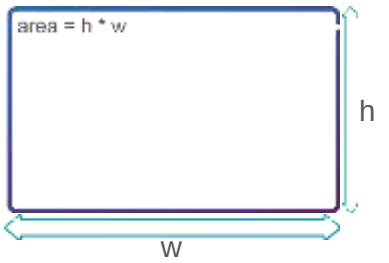
Access specifier

constructor

methods

members

```
class Rectangle
{
  public:
    Rectangle(double, double);
    double area(void);
    void display(void);
  private:
    double _w;
    double _h;
};
```



BASICS OF OBJECT ORIENTED PROGRAMMING

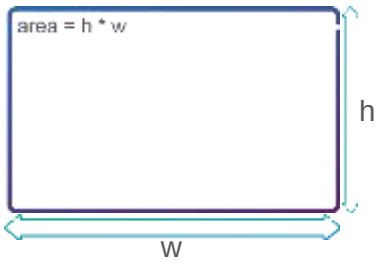
Class Definition (Rectangle.cpp)

```
Rectangle::Rectangle(double w, double h)
{
  _w = w;
  _h = h;
}
```

```
double Rectangle::area(void)
{
  return _w * _h;
}
```

```
void Rectangle::display(void)
{
  cout << "Width:  " << _w << endl;
  cout << "Height: " << _h << endl;
}
```

scope operator



BASICS OF OBJECT ORIENTED PROGRAMMING

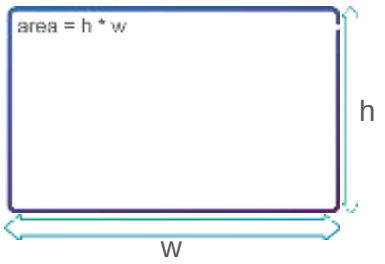
Usage (`main.cpp`) is very similar to structure usage.

```
#include <iostream>
#include "Rectangle.h"

using namespace std;

int main()
{
  Rectangle rect(2, 3.5);

  rect.display();
  cout << "Area = " << rect.area() << endl;

  return 0;
}
```

include Rectangle declaration

constructor _w = 2 _h = 3.5

method call



BASICS OF OBJECT ORIENTED PROGRAMMING

Access Specifier (encapsulation) sets the rights to access the variables and methods.

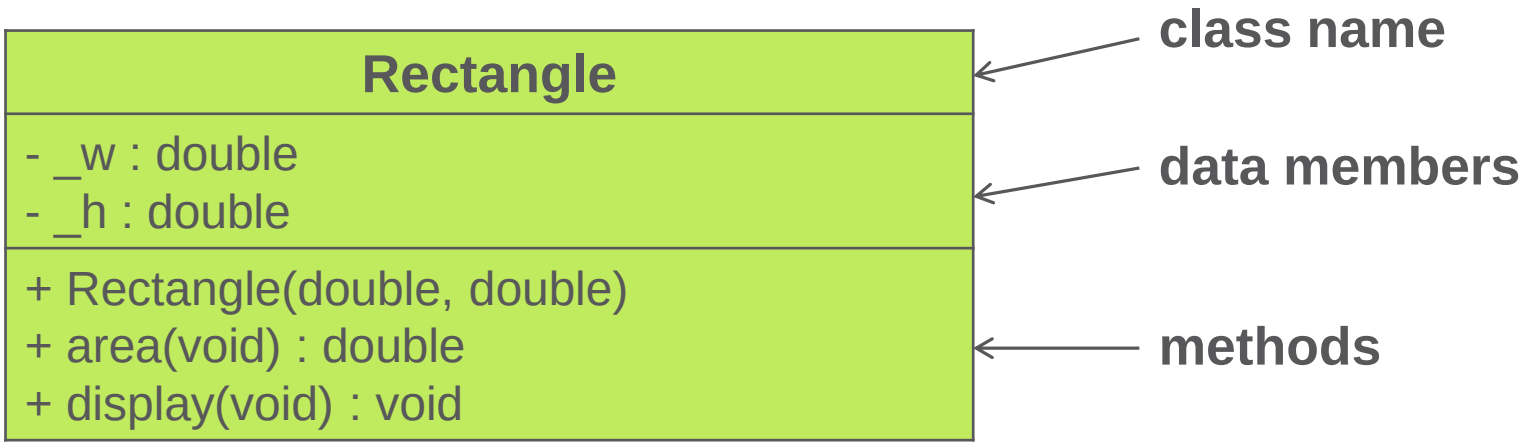
`[+]` **`PUBLIC`** – accessible from anywhere where the object is visible (default for structure)

`[-]` **`PRIVATE`** – accessible only from within the class or from class friends (default for class)

`[#]` **`PROTECTED`** – same as private with the exception that they can be accessed from within the members of their derived classes

BASICS OF OBJECT ORIENTED PROGRAMMING

Class diagram from the Unified Modeling Language provides the easy way of representing a class declaration.



BASICS OF OBJECT ORIENTED PROGRAMMING

Methods can be defined either inside the class or outside of it.

```
class Rectangle {
  public:
    Rectangle(double, double);
    double area() {
      return _w * _h;
    }
  private:
    double _w, _h;
};

Rectangle::Rectangle(double w, double h) {
  _w = w;
  _h = h;
}
```

declaration

declaration & definition

definition

Method defined inside the class will be treated as inline function.

BASICS OF OBJECT ORIENTED PROGRAMMING

Constructor is a special method used to initialize members or assign dynamic memory during the object creation.

```
Rectangle r(2, 3.5);
cout << "Area of rectangle = " << r.area() << endl;
```

constructor call

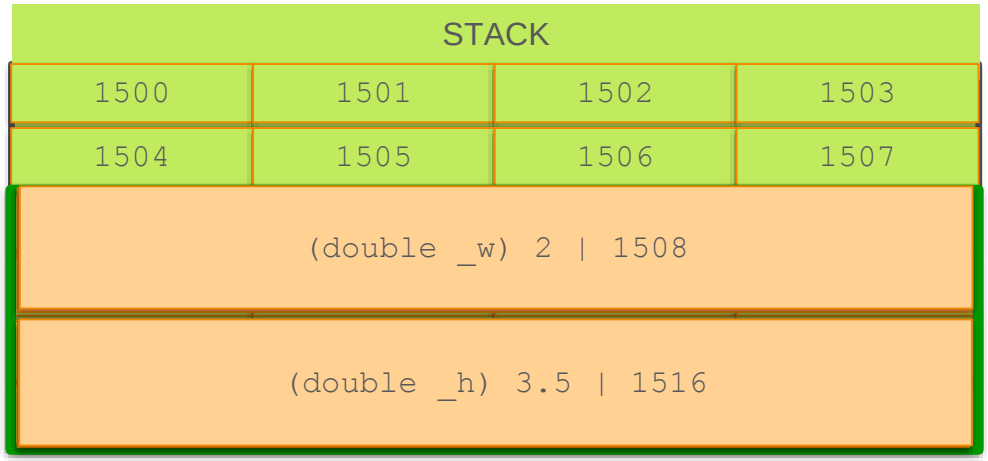
Constructor is a method that:

- Is automatically called when the object is created
- Has the same name as the `class`
- Returns no type (not even a `void`)
- Cannot be called explicitly

BASICS OF OBJECT ORIENTED PROGRAMMING

Constructor at work.

```
Rectangle::Rectangle(double w, double h)
{
  _w = w;
  _h = h;
}

Rectangle rec(2, 3.5);
```



1. **Allocate memory**
2. **Assign values**

BASICS OF OBJECT ORIENTED PROGRAMMING

Initialization List allows for initialization of members during memory allocation.

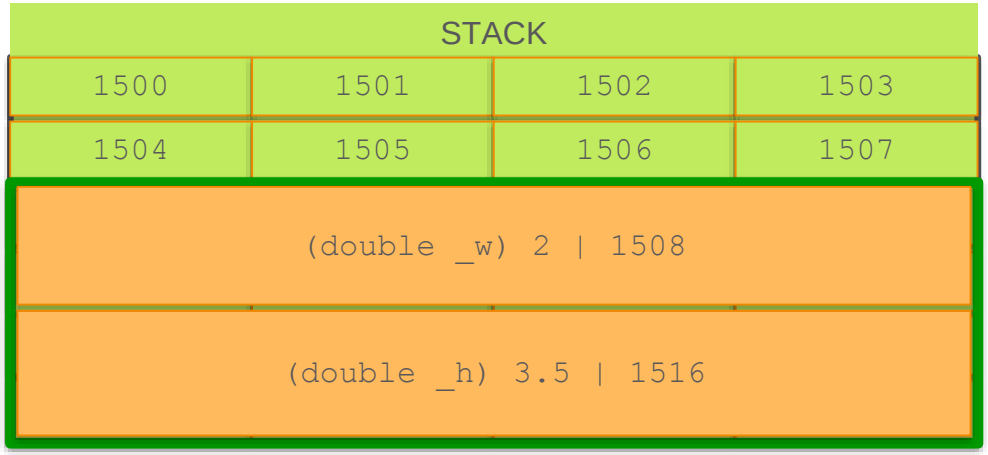
- Quicker than assigning values in constructor
- Order must be the same as in class declaration
- Constant members can only be set using initialization list
- Is very handy when it comes to inheritance

BASICS OF OBJECT ORIENTED PROGRAMMING

Constructor at work (initialization list).

```
Rectangle::Rectangle(double w, double h) :
  _w(w),
  _h(h)
{
  // nothing else to do
}

Rectangle rec(2, 3.5);
```

Initialization list



1. **Allocate memory with values**

BASICS OF OBJECT ORIENTED PROGRAMMING

Default Constructor is a special constructor that takes no arguments and is provided by class if no other constructor is defined.

```
class Rectangle
{
  public:
    double area(void);
    void display(void);
  private:
    double _w;
    double _h;
};
```

no constructor declared

(default constructor will be provided by compiler)

Default Constructor sets values of all members as if they were declared in the object scope (0 for global, unknown for local).

BASICS OF OBJECT ORIENTED PROGRAMMING

```
class Rectangle
{
  public:
    Rectangle(void);
    double area(void);
    void display(void);
  private:
    double _w;
    double _h;
};
```

user defined constructor with no parameters

(this is not the default constructor)

When using constructor with no parameters (default or user defined) remember not to use brackets “()”.

`Rectangle instance;`	Correct
`Rectangle function();`	Wrong

BASICS OF OBJECT ORIENTED PROGRAMMING

Class can have multiple constructors as long as they are distinguishable (have different number of arguments or the arguments are of different types – just like with the regular functions).

```
class Rectangle
{
  public:
    Rectangle(void);
    Rectangle(double, double = 2);
    double area(void);
    void display(void);
  private:
    double _w;
    double _h;
};
```

constructor #1
(no parameters)

constructor #2
(two parameters, default value for parameter)

BASICS OF OBJECT ORIENTED PROGRAMMING

Copy Constructor is a special constructor that allow the creation of an object using an already existing object of the same class.

```
Rectangle ( const Rectangle & );
```

```
Rectangle::Rectangle (const Rectangle &original)
{
  _w = original._w;
  _h = original._h;
}

Rectangle A(2. 3,5);
Rectangle B(A);
```

Rectangle A
- _w : double = 2 - _h : double = 3.5

Rectangle B
- _w : double = 2 - _h : double = 3.5

BASICS OF OBJECT ORIENTED PROGRAMMING

There are several signatures that are considered to be copy constructor:

```
<class name> (const <class name>
&);

<class name> (<class name> &);
```

If you do not declare a copy constructor, the compiler will generate a **default copy constructor** that:

- Copy all the members values using initialization list

```
Rectangle::Rectangle (const Rectangle &other) :
  _w(other._w),
  _h(other._h)
{ }
```

BASICS OF OBJECT ORIENTED PROGRAMMING

Objects creation and destruction.

```
#include <iostream>
#include "Rectangle.h"

int main()
{
  Rectangle r(2, 3.5);
  Rectangle *p = new Rectangle;
  {
    Rectangle clone(r);
  }
  delete p;
}
```

Object is destroyed when:

- Its scope has ended
- Is explicitly destroyed

BASICS OF OBJECT ORIENTED PROGRAMMING

Destructor is a special method that fulfils the opposite functionality to the constructor:

- It is called automatically when the object is destroyed
- Must have the same name as the class, but preceded by a ~ (tilde) character
- Returns no type (not even a `void`)
- Takes no parameters
- Only one destructor per class

destructor

```
Rectangle::~Rectangle()
{
    cout << "Destructor working." << endl;
}
```

BASICS OF OBJECT ORIENTED PROGRAMMING

```
class Memory
{
  public:
    Memory(const int & size);
    ~Memory();
  private:
    int *p;
};
```

constructor

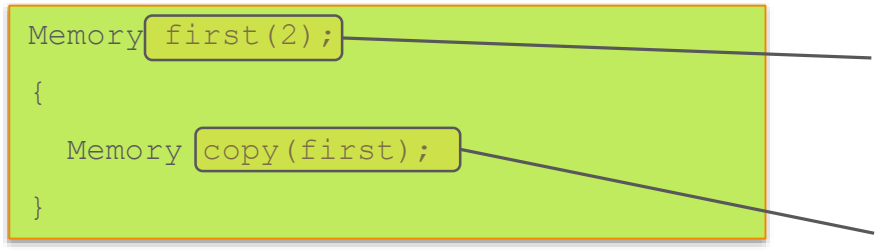
(will allocate memory)

destructor

(will free memory)

```
Memory::Memory(const int & size)
{
  p = new int[size];
}

Memory::~Memory()
{
  delete[] p;
}
```

BASICS OF OBJECT ORIENTED PROGRAMMING

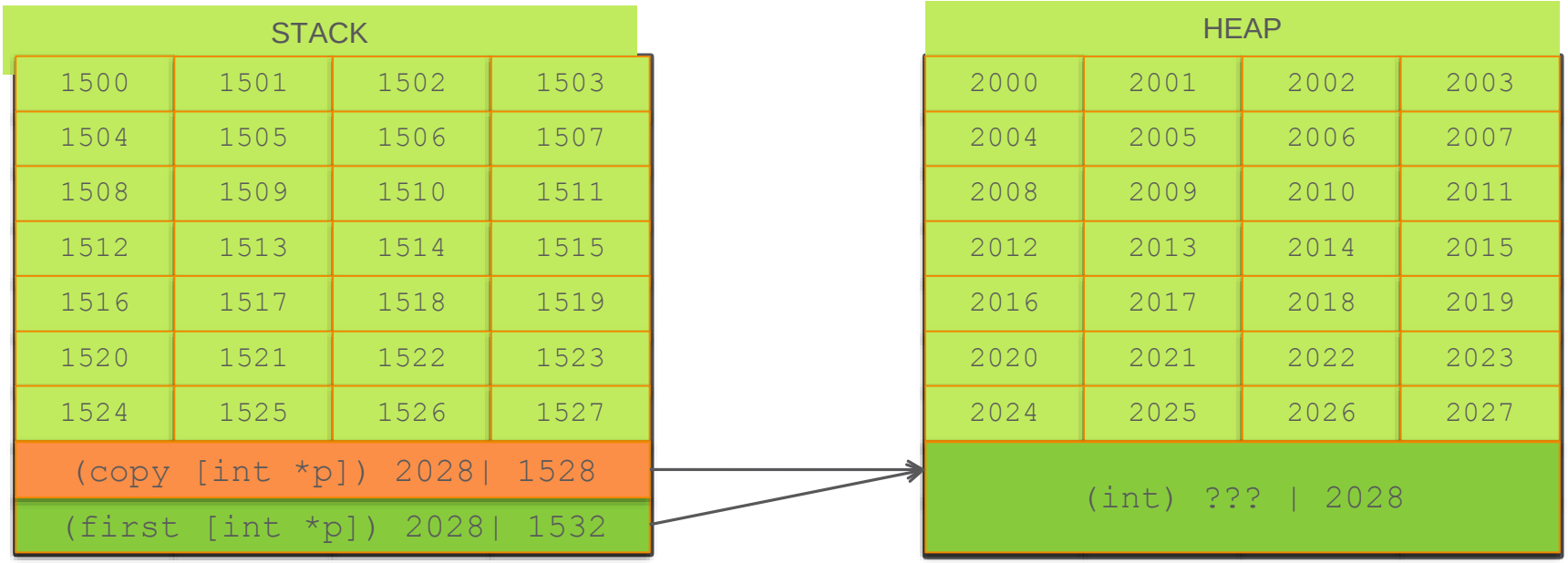


first – global variable

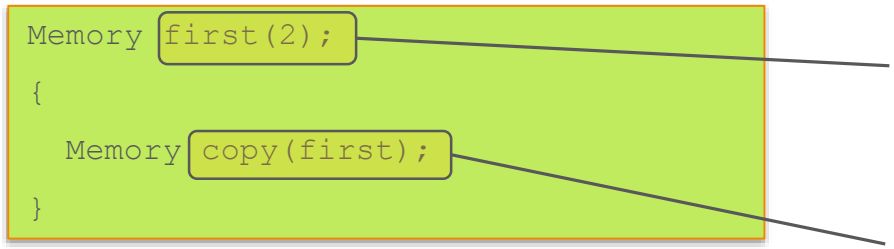
(will allocate 8 bytes)

copy – local variable

(will copy values of first)



BASICS OF OBJECT ORIENTED PROGRAMMING

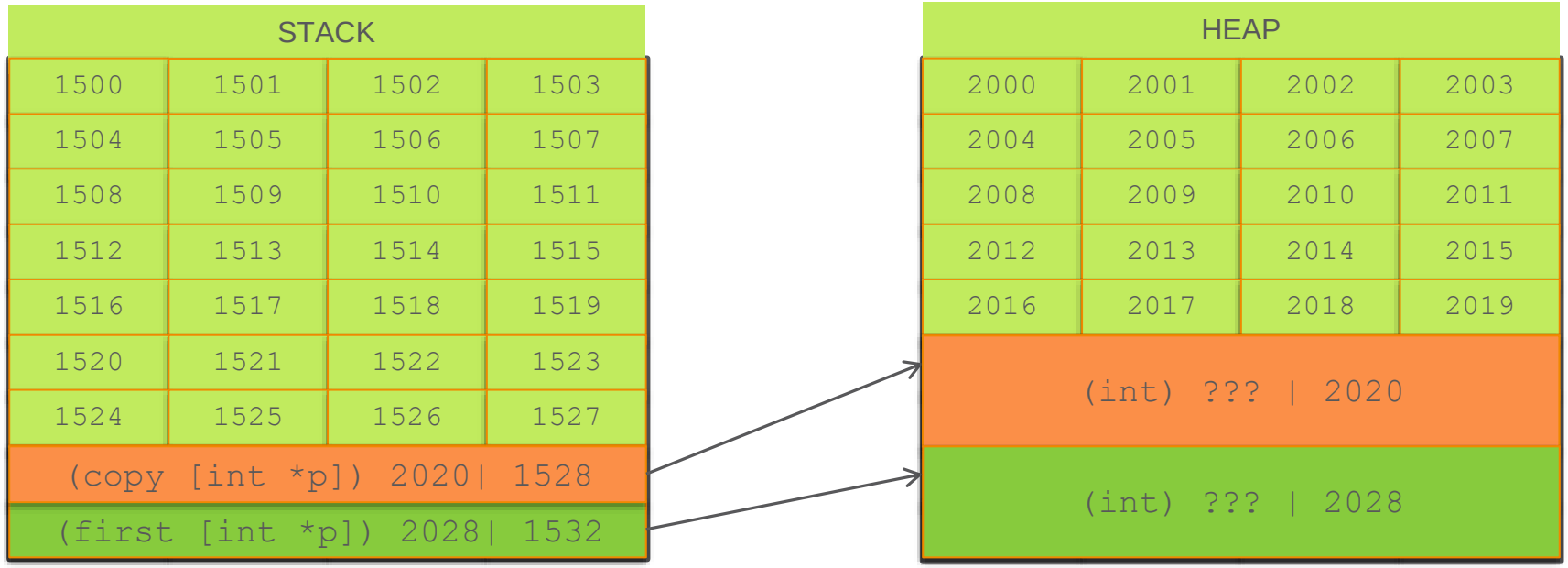


first – global variable

(will allocate 8 bytes)

copy – local variable

(will copy values of first)



BASICS OF OBJECT ORIENTED PROGRAMMING

```
class Memory
{
  public:
    Memory(const int & size);
    ~Memory();
    Memory(const Memory &);
  private:
    int _size;
    int *p;
};
```

copy constructor

(my own = better)

size of the array

(will be used for copying)

```
Memory::Memory(const int & size) : _size(size) {
  p = new int[size];
}

Memory::~Memory() {
  delete[] p;
}
```

BASICS OF OBJECT ORIENTED PROGRAMMING

```
#include <cstring>


Memory::Memory(const Memory & other) :
  _size(other._size)
{
  p = new int[_size];
  memcpy(p, other.p, _size * sizeof(int));
};
```

allocation

(allocate new memory on heap)

copy the content of heap of the original

(destination, source, how many bytes)

BASICS OF OBJECT ORIENTED PROGRAMMING

Pointer to a class may be defined the same way as for any basic variable type or structure. The class elements are accessed the same way as structures, by using −> (arrow) operator.

```
void main()
{
  Rectangle *p = new Rectangle(2, 3.5);
  cout << "Area = " << p->area() << endl;
  delete p;
}
```

access via pointer

(operator "->")

BASICS OF OBJECT ORIENTED PROGRAMMING

Operator overloading enables the definition of operators in the class**.**

```
class Vector3D
{
  public:
    Vector3D(double, double, double);
  private:
    double _dx;
    double _dy;
    double _dz;
};

Vector3D::Vector3D (double dx, double dy, double dz) :
  _dx(dx),
  _dy(dy),
  _dz(dz)
{ }
```

BASICS OF OBJECT ORIENTED PROGRAMMING

Compiler does not know how to add Vector3D objects.

```
void main()
{
  Vector3D A(1, 2, 3);
  Vector3D B(3, 4, 5);

  Vector3D C = A + B;
}
```

We would like it to be:

(4, 6, 8)

Operator + must be defined for class Vector3D.

BASICS OF OBJECT ORIENTED PROGRAMMING

Adding Vector3D.

```
Vector3D Vector3D::add(const Vector3D &v)
{
  Vector3D result(_dx, _dy, _dz);

  result._dx += v._dx;
  result._dy += v._dy;
  result._dz += v._dz;

  return result;
}
```

```
void main()
{
  Vector3D A(1, 2, 3);
  Vector3D B(3, 4, 5);

  Vector3D C = A.add(B);
}
```

method param
(Vector3D to add)

method name
(add)

result of adding
(Vector3D)

result is “me
(_dx, _dy, _dz)

plus v”
(v._dx, v._dy, v._dz)

BASICS OF OBJECT ORIENTED PROGRAMMING

Adding Vector3D.

```
Vector3D Vector3D::operator+(const Vector3D &v)
{
  Vector3D result(_dx, _dy, _dz);

  result._dx += v._dx;
  result._dy += v._dy;
  result._dz += v._dz;

  return result;
}
```

```
void main()
{
  Vector3D A(1, 2, 3);
  Vector3D B(3, 4, 5);

  Vector3D C = A + B;
}
```

method name

(operator+)

or just “+”

BASICS OF OBJECT ORIENTED PROGRAMMING

The following operators can be overloaded:

Overloadable Operators										
+	-	*	/	=	<	>	+=	-=	*=	/=
<<	>>	<<=	>>=	==	!=	<=	>=	++	--	%
&	^	!	|	~	&=	^=	|=	&&	||	%=
[]	()	,	->*	->	new	delete	new[]	delete[]		

The only operator overloaded by default is “=” (assignment operator):

- One argument: object of the class type
- Copy each class member by value

```
Vector A, B;
B = A;
```

BASICS OF OBJECT ORIENTED PROGRAMMING

Class can contain `static` members:

- Are called “class members” – they are not bound to the instance, rather to the class itself
- Have the same properties as global

```
class Counter
{
  public:
    unsigned int id;
    static int count;
    Counter() { count++; }
    ~Counter() { count--; }
};

int Counter::count = 0;
```

static variable
(declaration)

static variable
(definition)

BASICS OF OBJECT ORIENTED PROGRAMMING

There is only one memory area where the static variable is held.

```
int main()
{
  Counter first;
  Counter *second = new Counter;
  Counter third;
}
```

STACK

([static] int count) 3 | 1500			
1504	1505	1506	1507
1508	1509	1510	1511
(Counter third) ? | 1512			
(Counter *second) ? | 1516			
(Counter first) ? | 1520			

BASICS OF OBJECT ORIENTED PROGRAMMING

Accessing static members can be achieved by:

- The class name and scope operator (::)
- Using the instance of the class

```
int main()
{
  cout << Counter::count << endl;

  Counter first;
  count << first.count << endl;

  Counter *second = new Counter;
  cout << second->count << endl;

  return 0;
}
```

<class>::<variable>

(by class)

<instance>.<variable>

(by instance)

<pointer>-><variable>

(by pointer)

BASICS OF OBJECT ORIENTED PROGRAMMING

Methods also can be `static`:

› Can operate only on `static` members

```
class Counter
{
  public:
    unsigned int id;
    static int getCount()
    {
      return _count;
    }

    Counter() { _count++; }
    ~Counter() { _count--; }
  private:
    static int _count;
};

int Counter::_count = 0;
```

static method

(_count is also static)

BASICS OF OBJECT ORIENTED PROGRAMMING

Accessing static methods can be achieved by exactly the same way as accessing static members.

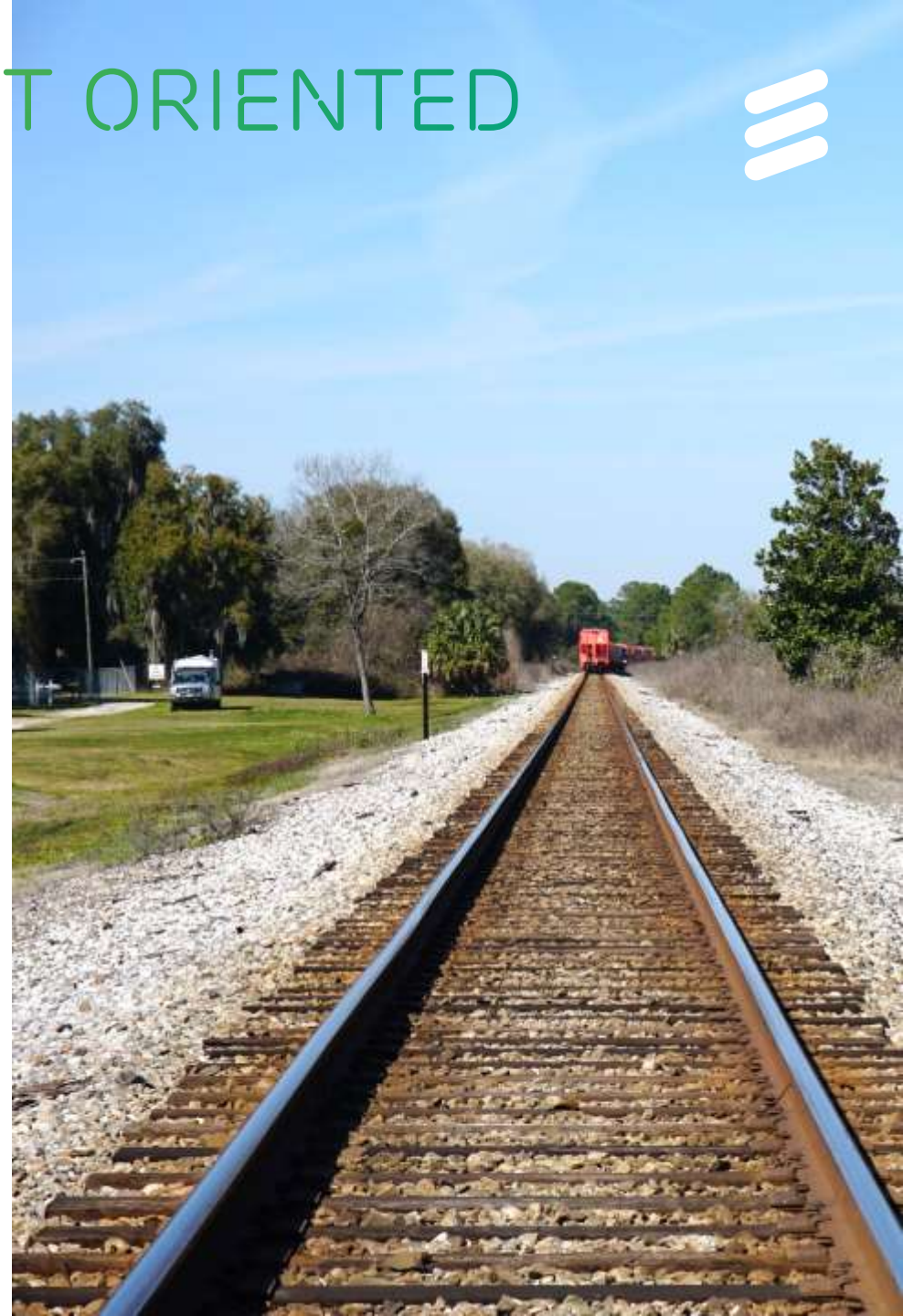
```
int main()
{
  cout << Counter::getCount() << endl;

  Counter first;
  count << first.getCount() << endl;

  Counter *second = new Counter;
  cout << second->getCount() << endl;

  return 0;
}
```

<class>::<method>
(by class)

<inst>.<method>
(by instance)

<ptr>-><method>
(by pointer)

BASICS OF OBJECT ORIENTED PROGRAMMING

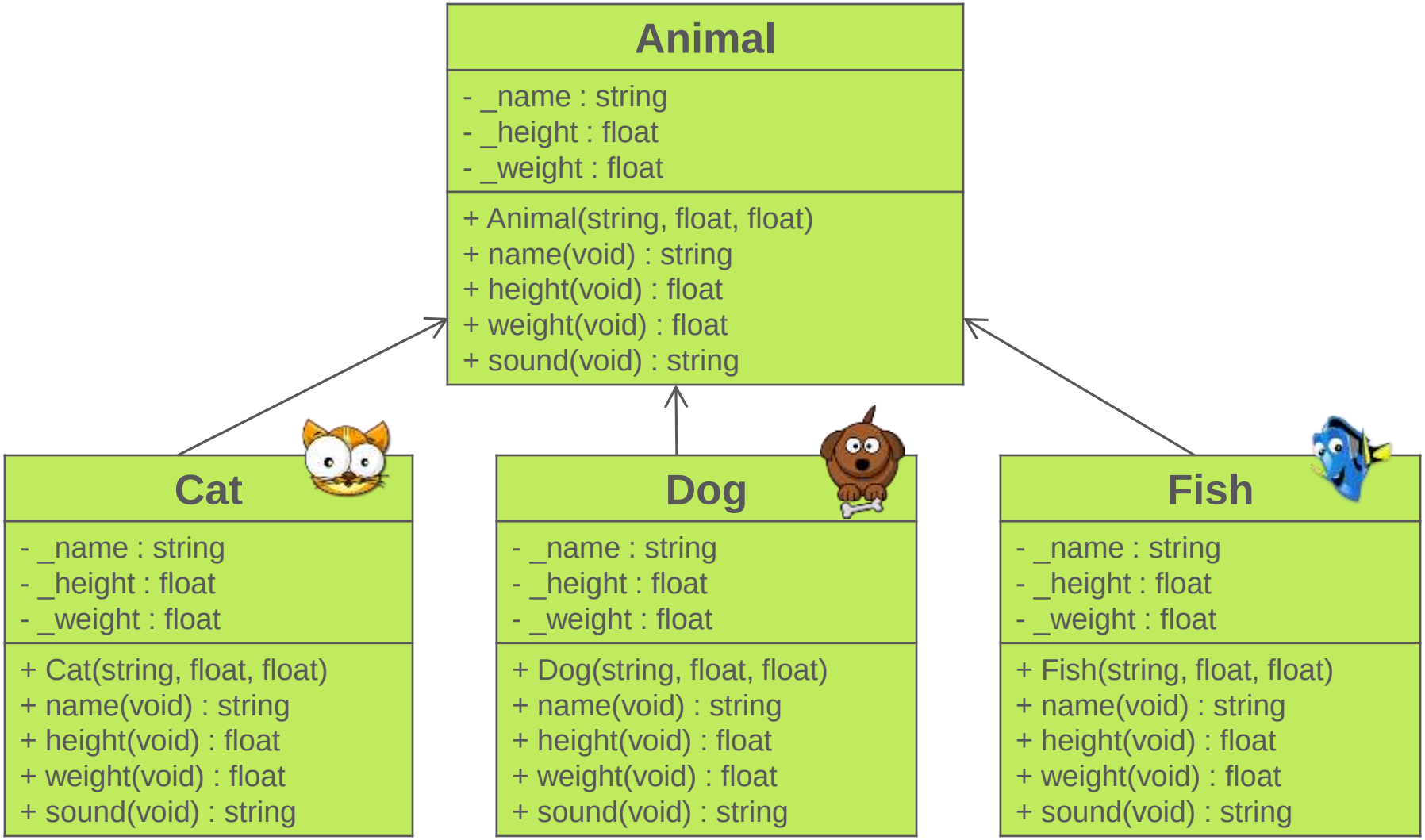


Summary:

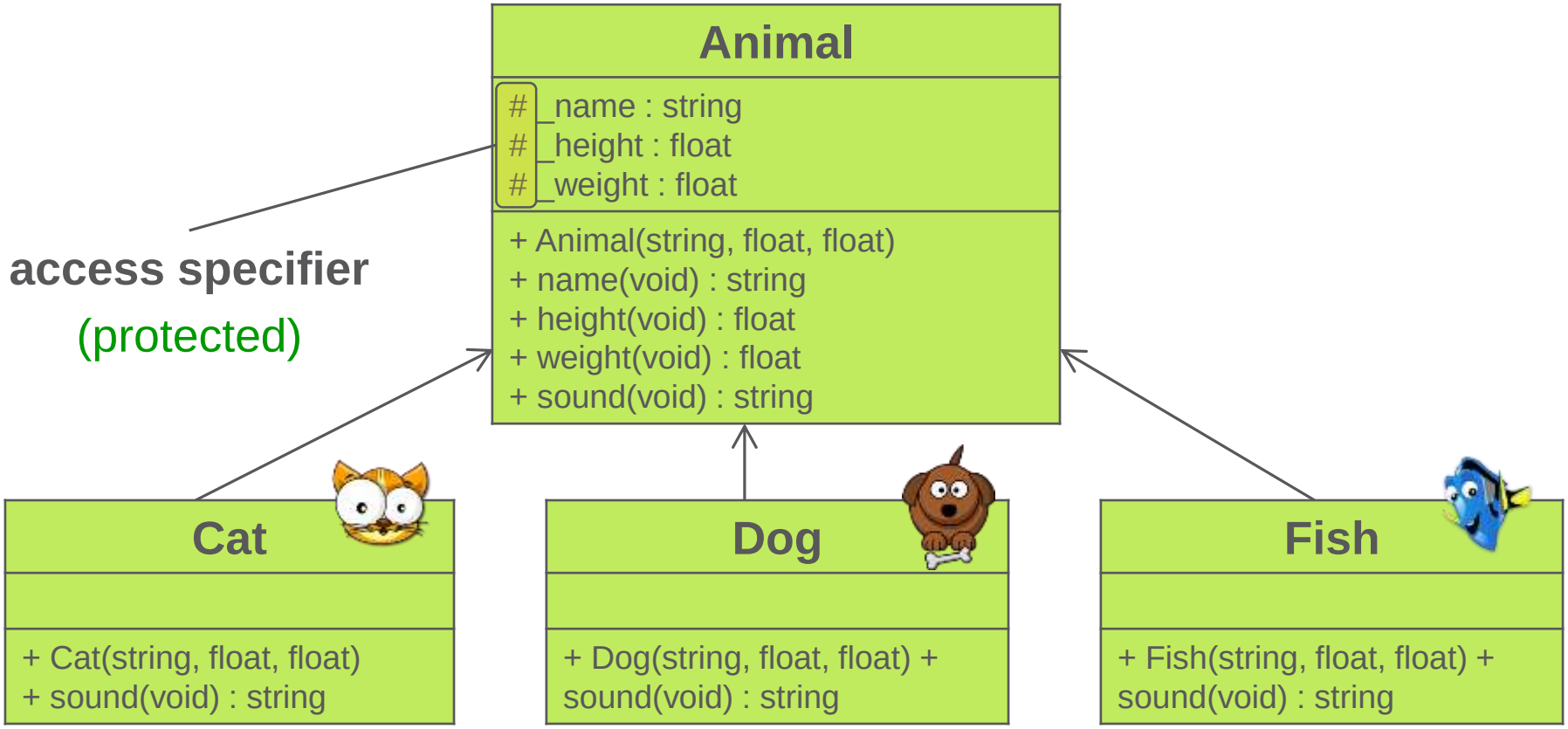
- **Class**
  - Members and methods
- **Construction**
  - Constructor
  - Constructors initialization list
  - Copy constructor
- **Destruction**
  - Destructor
- **Other**
  - Operators overloading
  - Static members and methods

CHAPTER II: INHERITANCE

INHERITANCE

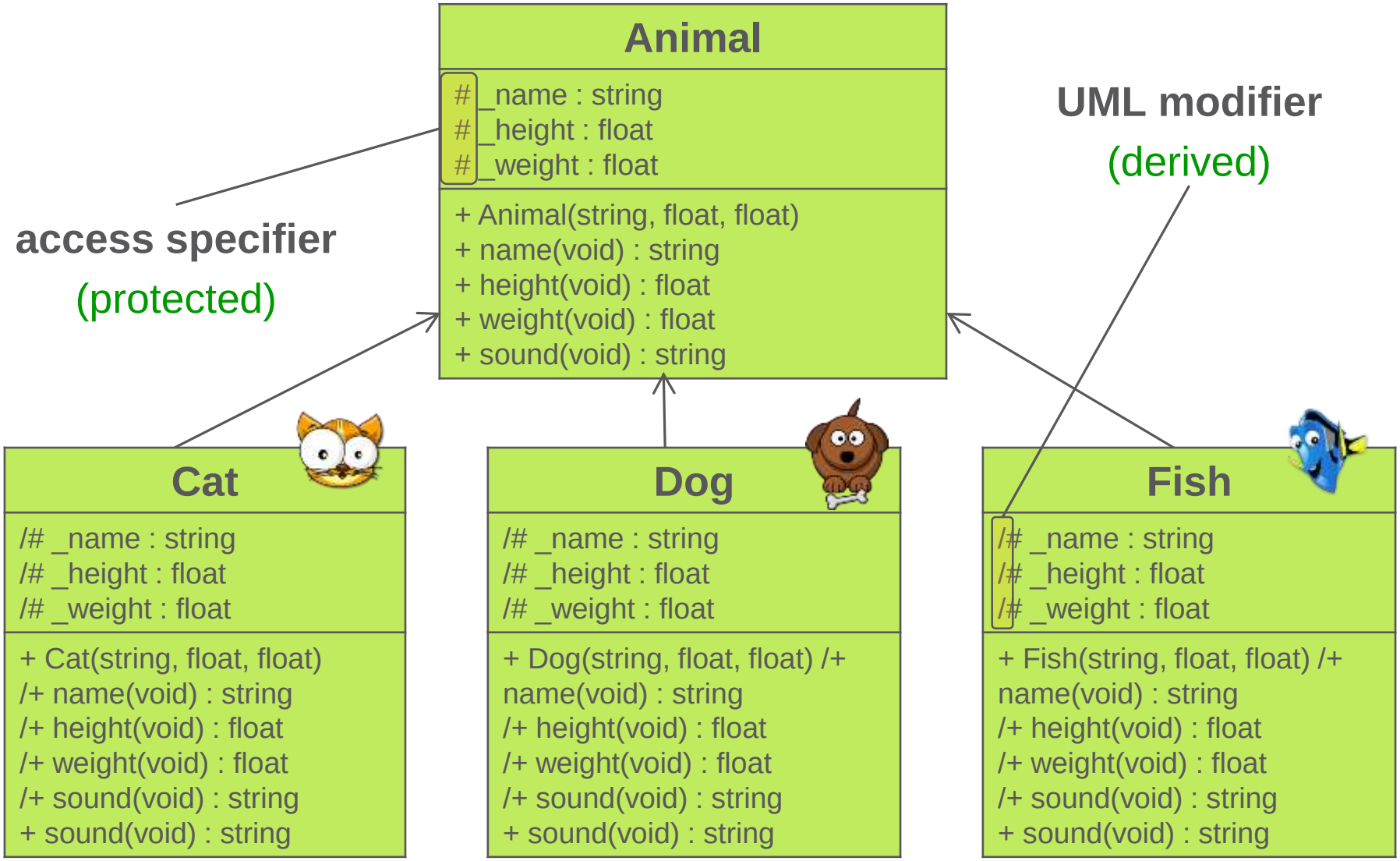


INHERITANCE



Inheritance allows to create classes which are automatically including some of its “parents” members.

INHERITANCE



INHERITANCE

Animal class.

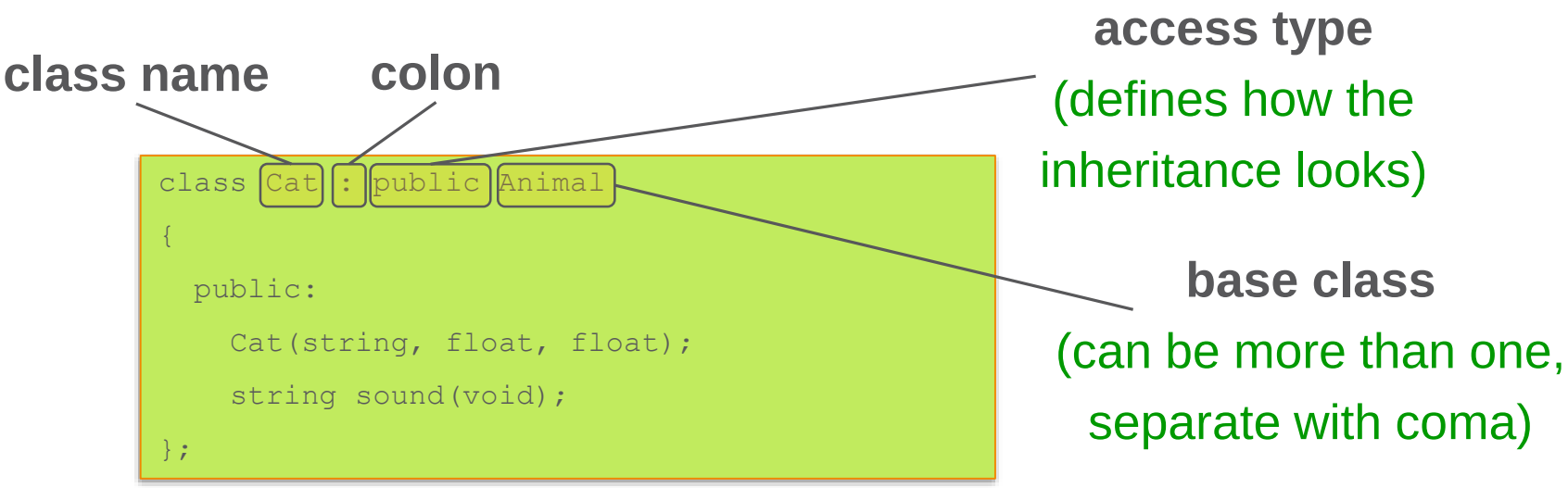
```
class Animal {
  public:
    Animal(string, float, float);
    string name() { return _name; }
    float height() { return _height; }
    float weight() { return _weight; }
    string sound(void);
  protected:
    string _name;
    float _height, _weight;
};
```

```
Animal::Animal(string name, float height, float weight) :
  _name(name), _height(height), _weight(weight)
{ }
```

```
string Animal::sound(void)
{
  return "Unknown sound or no sound";
}
```

INHERITANCE

How to derive?



Access type specifies how the inheritance should look like. The possible access type identifiers are:

- public
- protected
- private

INHERITANCE

The difference between access types can be summarized according to who can access them:

Access	public	protected	private
Members of the same class	✓	✓	✓
Members of derived class	✓	✓	✗
Not members	✓	✗	✗

Base class member access specifier	Type of inheritance		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	n/a Hidden	n/a Hidden	n/a Hidden

INHERITANCE

Class Cat (Dog and Fish)**.**

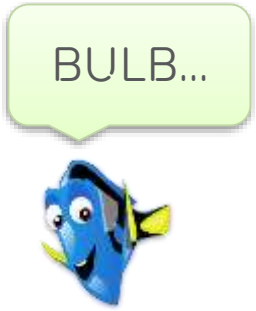
```
class Cat : public Animal
{
  public:
    Cat(string, float, float);
    string sound(void);
};
```

reuse of code

(base class constructor)

```
Cat::Cat(string name, float height, float weight) :
  Animal(name, height, weight)
{ }
```

```
string Cat::sound(void)
{
  return "Meow...";
}
```



INHERITANCE

```
int main()
{
  Cat garfield("Garfield", 10, 50);
  Fish nemo("Nemo", 0.5, 1);

  cout << garfield.name() << ":" endl;
  cout << garfield.height() << endl;
  cout << garfield.weight() << endl;
  cout << garfield.sound() << endl;

  cout << nemo.name() << ":" endl;
  cout << nemo.height() << endl;
  cout << nemo.weight() << endl;
  cout << nemo.sound() << endl;
  cout << nemo.Animal::sound() << endl;

  return 0;
}
```

derived
(from Animal)

not derived
(from Cat)

not derived
(directly from Animal)

INHERITANCE

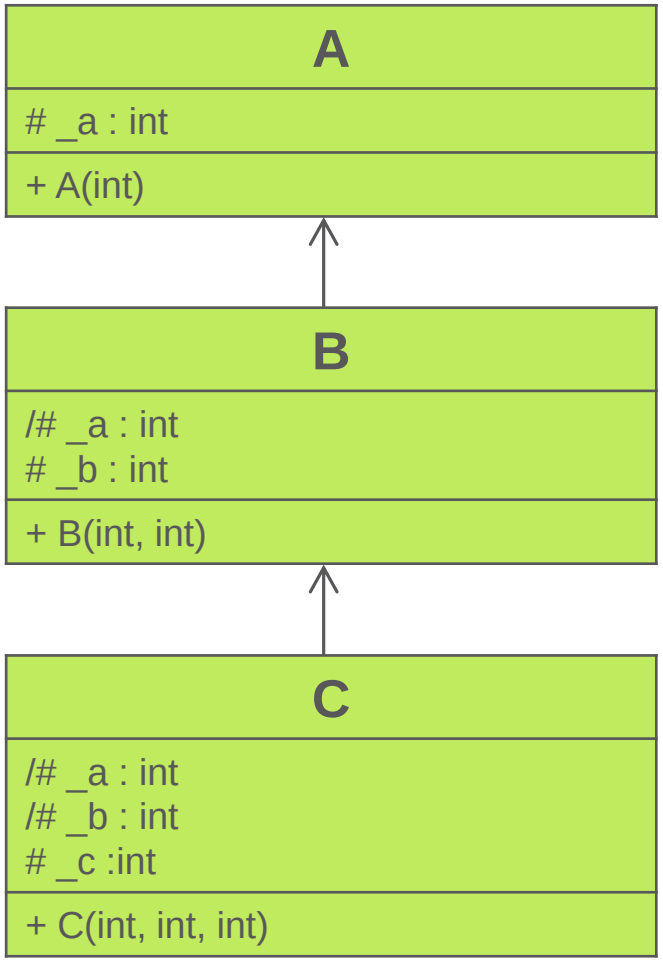
Everything is inherited but:

- Constructor and destructor
- Operator “=”
- Base class friends

Do you want to know more?

http://www.cplusplus.com/doc/tutorial/inheritance/

INHERITANCE



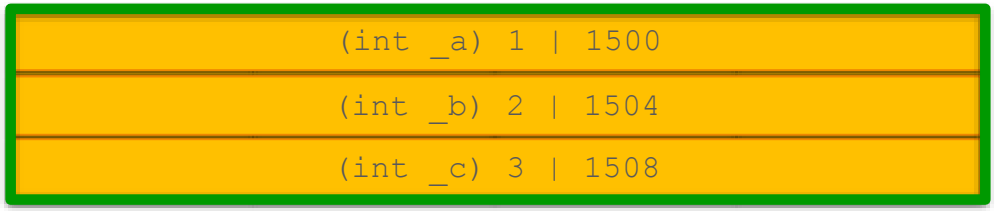

```
A::A(int a) :
  _a(a)
{ cout << "A!" << endl; }
```

```
B::B(int a, int b) :
  A(a), _b(b)
{ cout << "B!" << endl; }
```

```
C::C(int a, int b, int c) :
  B(a, b), _c(c)
{ cout << "C!" << endl;}
```

```
C objectC(1, 2, 3);
```

STACK



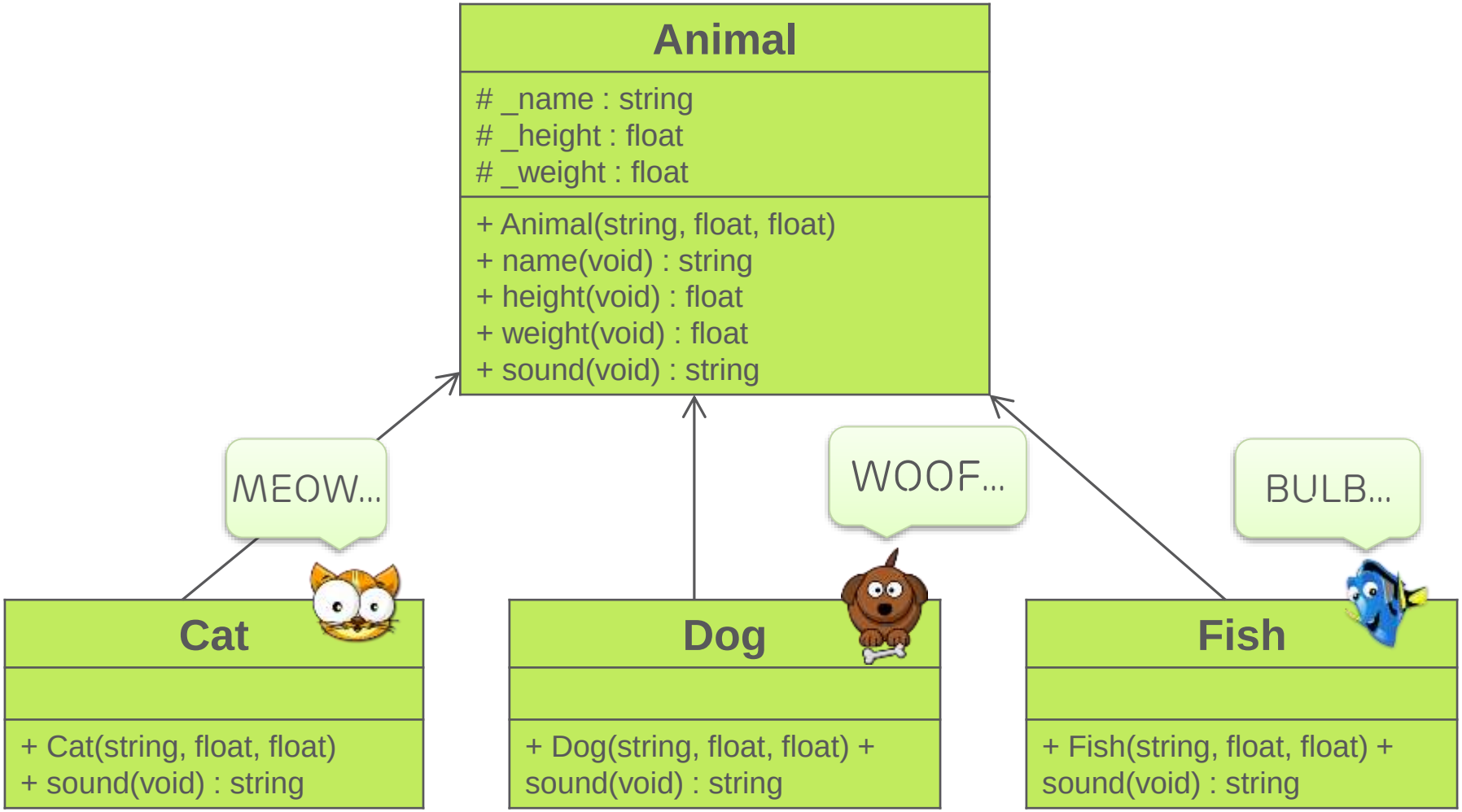
INHERITANCE

Summary:

- **Types of inheritance**
  - Public
  - Protected
  - Private
- **Constructors call order**
  - Up to bottom
- **Destructions call order**
  - Bottom to up

CHAPTER III: POLYMORPHISM

POLYMORPHISM



POLYMORPHISM

Which combinations are allowed?

nothing special

(normal creation of a Cat)

```
Cat cat("Garfield", 10, 0.5);

Animal &animalReference = cat;

Animal *animalPointer = &cat;
```

reference

(Cat is being referenced as Animal)

pointer

(Cat is being pointed by Animal pointer)

POLYMORPHISM

Which combinations are allowed?

function

(Animal received by value)

```
void fun(Animal animal);

Cat cat("Garfield", 10, 0.5);
fun(cat);
```

Nothing special

(normal creation of a Cat)

function call

(with Cat as an argument)

POLYMORPHISM

The following combinations are allowed:

- Objects of the derived class can be referenced using Base class
- Objects of the derived class can be pointed using pointers to the Base class
- Objects of the derived class can be send to the function as a parameter of Base class (by value or reference)

HOWEVER…

POLYMORPHISM

Derived class object treated as a base class object behaves as one (base class object).

```
Cat cat("Garfield", 10, 0.5);

Animal &animalReference = cat;
Animal *animalPointer = &cat;

cout << animalReference.sound() << endl;
cout << animalPointer->sound() << endl;
```

treated as Animal
(sound() will be taken from Animal)

UNKNOWN SOUND OR NO SOUND

Slicing: while using pointer or reference of base class, only methods and variables from the base class are available.

POLYMORPHISM

```
class Animal {
  public:
    Animal(string, float, float);
    string name() { return _name; }
    float height() { return _height; }
    float weight() { return _weight; }
    virtual string sound(void);
  protected:
    string _name;
    float _height, _weight;
};
```

polymorphism

(ON)

```
Cat cat("Garfield", 10, 0.5);

Animal &animalReference = cat;
Animal *animalPointer = &cat;

cout << animalReference.sound() << endl;
cout << animalPointer->sound() << endl;
```

this is Cat

(polymorphism will take this into account)

MEOW...

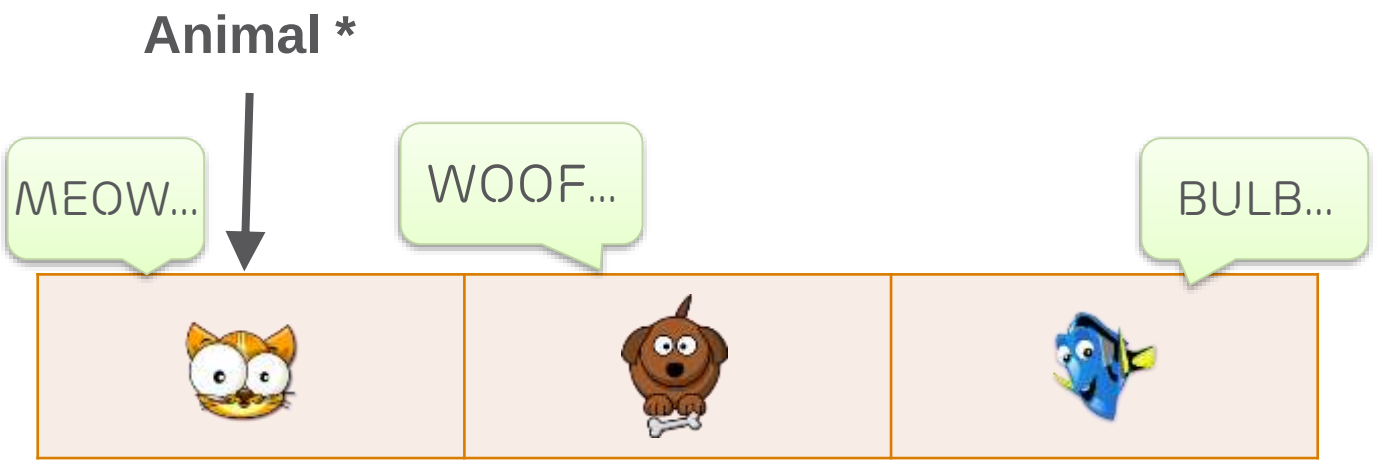
POLYMORPHISM

Polymorphism means having multiple forms. Single method (and hence the whole class) can be made polymorphic by using `virtual` keyword.

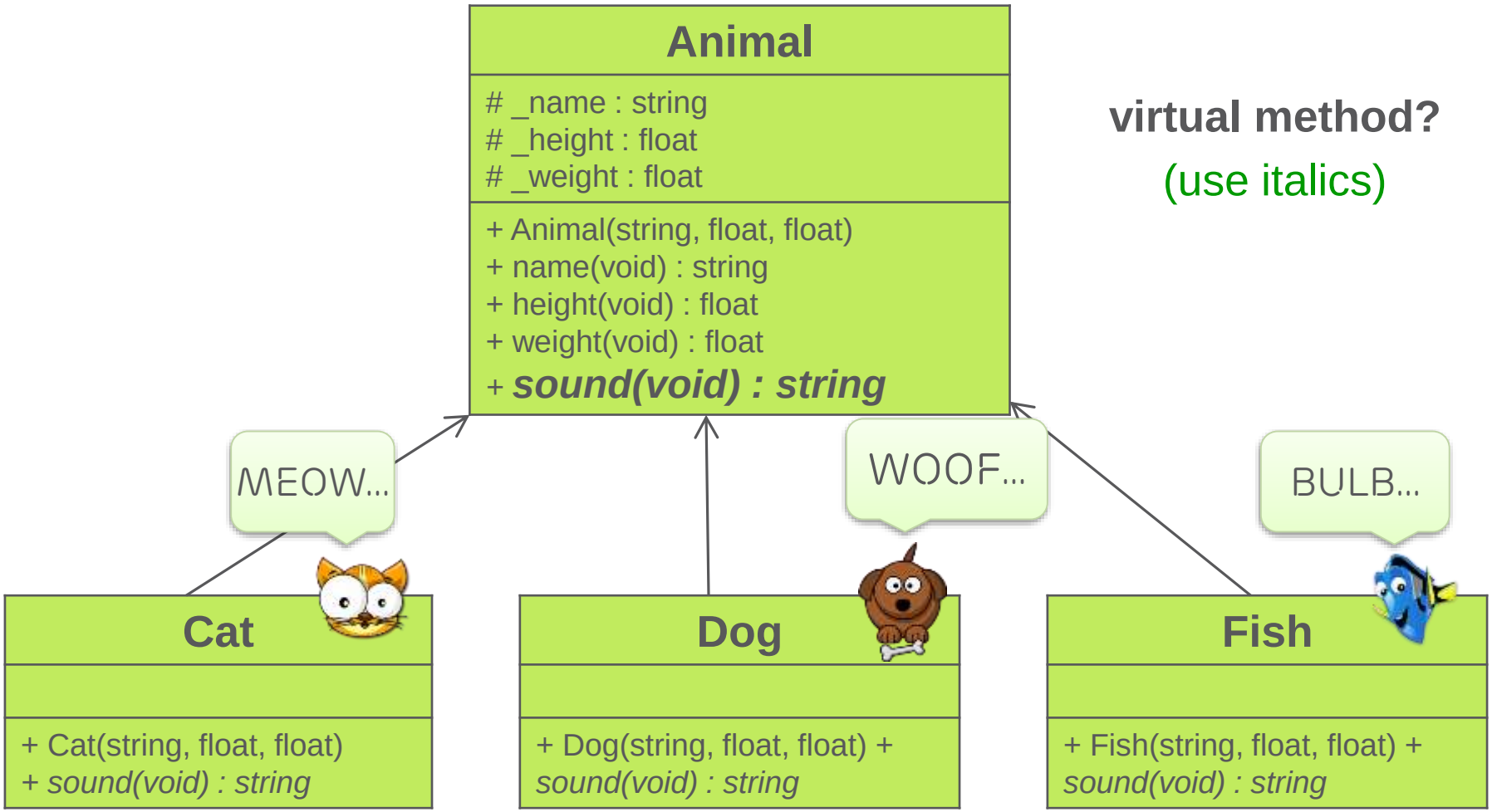
- Constructor may never be virtual
- Destructor of a polymorphic class must be virtual
- Polymorphic method is automatically polymorphic in all derived classes
- When calling virtual method on an instance using pointer or reference to a base class, the method from derived class is called (if defined)

POLYMORPHISM

```
Animal *zoo[3];
zoo[0] = new Cat("Garfield", 10, 1);
zoo[1] = new Dog("Pluto", 15, 2);
zoo[2] = new Fish("Nemo", 3, 0.2);

for (int i = 0; i < 3; ++i)
{
  cout << zoo[i]->sound() << endl;
}
```



POLYMORPHISM



POLYMORPHISM

What is Animal?

- Cat
- Dog
- Fish
- Frog
- etc…

The same question goes for Figure, Vehicle, Chemical Element and many more.

These are examples of **Abstract** concepts that require CONCRETIZATION.

POLYMORPHISM

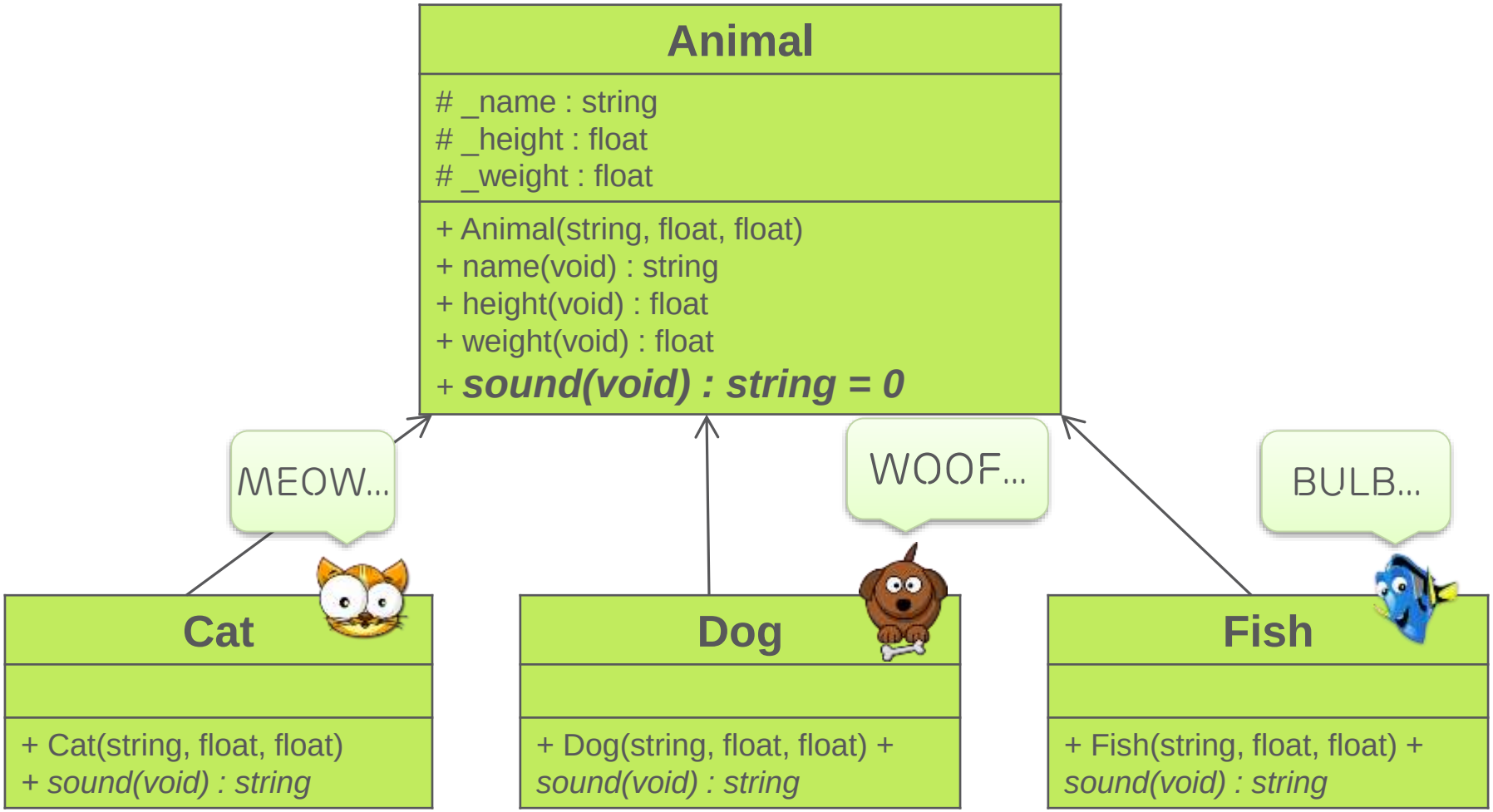
Abstract class is a class with at least one pure virtual method.

Pure virtual method declaration:

```
virtual <return type> <name> (<parameters>) = 0;
```

- Pure virtual method has no body
- No instance of abstract class can be created
- In every derived class, each pure virtual method of base class must be overloaded

POLYMORPHISM



POLYMORPHISM

```
class Animal {
  public:
    Animal(string, float, float);
    string name() { return _name; }
    float height() { return _height; }
    float weight() { return _weight; }
    virtual string sound(void) = 0;
  protected:
    string _name;
    float _height, _weight;
};
```

pure virtual method
(makes the class ABSTRACT)

```
Animal bug("Ant", 0.01, 0.02);
```

instance of Animal
(compilation error Animal is abstract)

```
Cat lazyCat("Garfield", 22, 5);
Animal *p = &lazyCat;

cout << p->sound() << endl;
```

still correct!

MEOW...

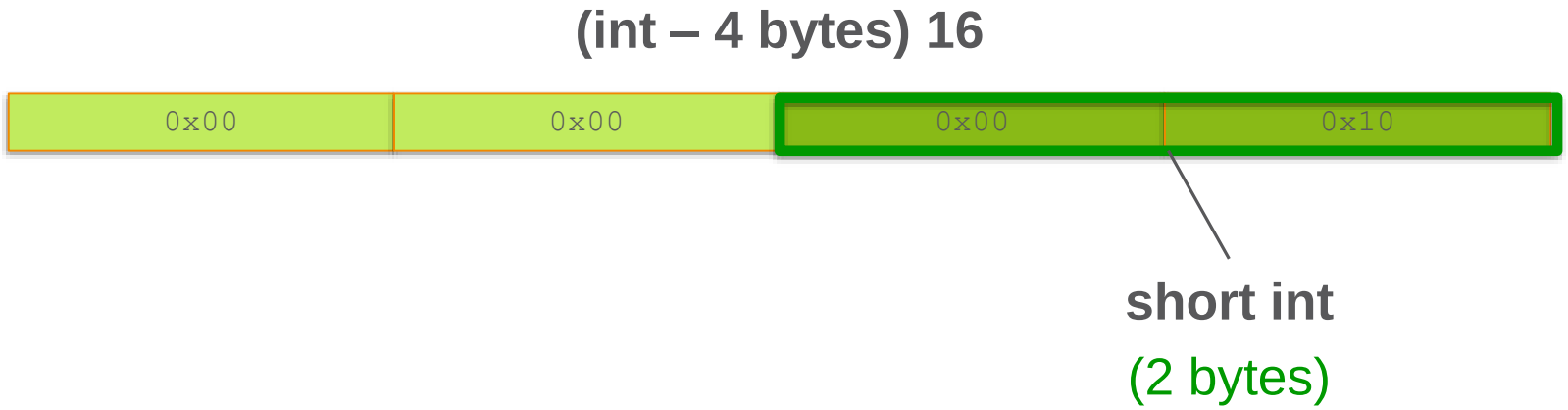
POLYMORPHISM

Summary:

- **Virtual methods**
  - Enables polymorphism (multiple forms)
  - Pure virtual methods
- **Abstract class**
  - At least one pure virtual method
  - No instance can be created

CHAPTER IV: TYPE CASTING

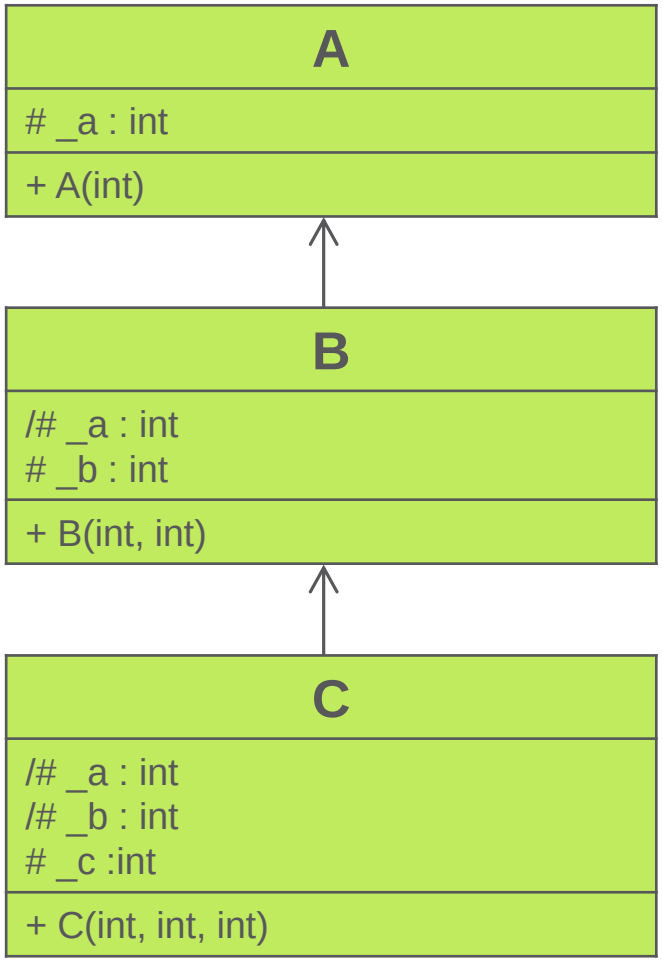
TYPE CASTING



C++ introduces new format for explicit casting of build in types.

```
int a = 16;
short b;
b = (short)a;                  // C-like cast notation
b = short(a); // function like C++ notation
```

TYPE CASTING




```
A a
B b;

a = b;
```

implicit conversion

```
A *p = new C;
```

implicit conversion

how to make such conversion?

TYPE CASTING

Conversion between classes in C++ shall be handled using the following casting operators:

```
dynamic_cast <new type> (expression)
static_cast <new type> (expression)
reinterpret_cast <new type> (expression)
```

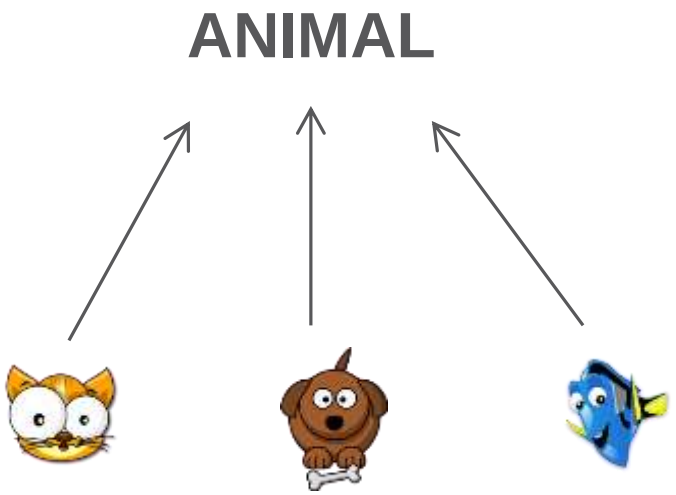
Constness in C++ can be added or removed on request:

```
const_cast <new type> (expression)
```

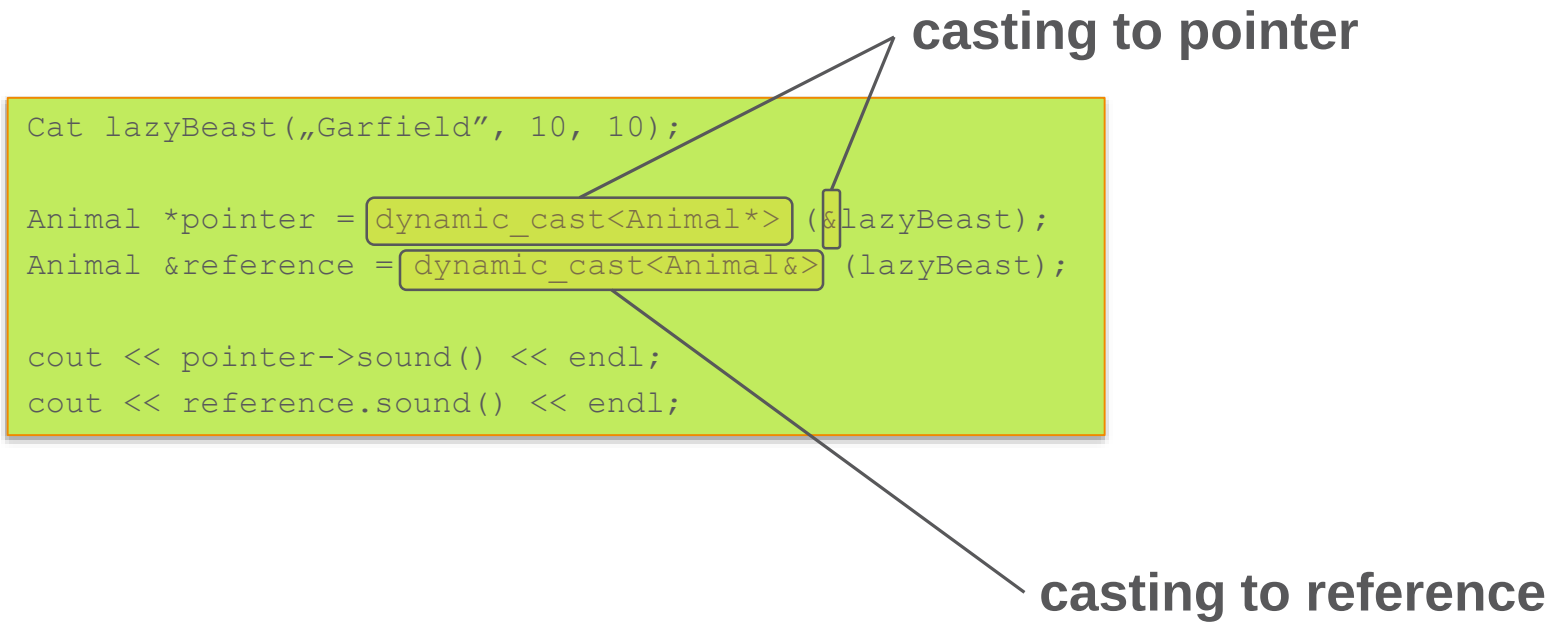
TYPE CASTING

Dynamic cast can only be used with pointers and references to objects.

- Ensures that the result is valid and complete object of requested class
- Successful when casting class to one of its base classes



TYPE CASTING



If casting to pointer is unsuccessful the `NULL` pointer is returned.

If casting to reference is unsuccessful exception is thrown.

TYPE CASTING

Static cast can perform conversions between pointers of related type.

- No safety checks are made
- Less overhead than `dynamic_cast`

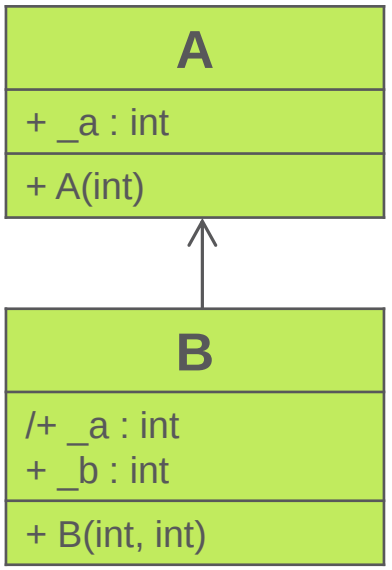
```
A *a_pointer = new A;
B *b_pointer = static_cast<B*> (a_pointer);
```

casting “down the ladder” (A to B)

```
B *b_pointer = new B;
A *a_pointer = static_cast<A*> (b_pointer);
```

casting “up the ladder”

(B to A – similar to dynamic_cast)



TYPE CASTING


```
short a = 5;
int b;
b = (int)a;          // C-like cast notation
b = int(a);          // function like C++ notation
```

Casting of build in types should be performed using `static_cast`.

```
double pi = 3.14159;
int i = static_cast<int>(pi);
```

TYPE CASTING

Reinterpret cast can perform conversions between pointers of any, even unrelated type.

- No safety checks are made
- Results in simple binary interpretation
- Should be used very carefully

=

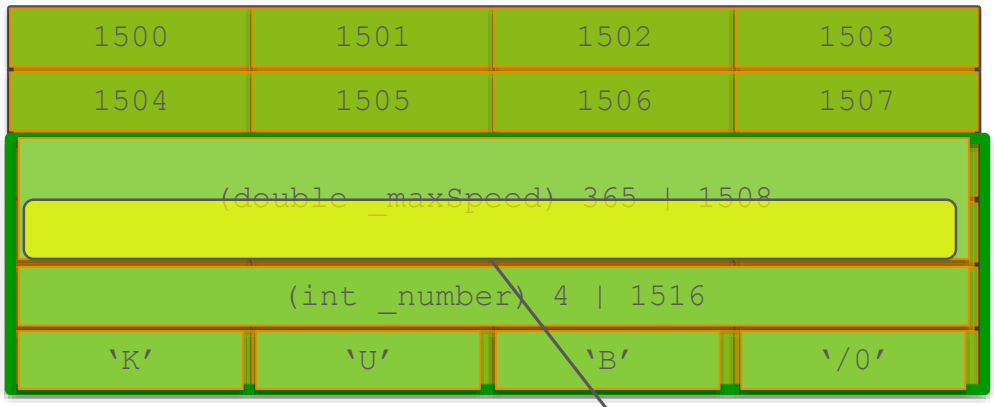
TYPE CASTING

Fish

+4

/# _name : string
/# _height : float
/# _weight : float

+ Fish(string, float, float) /+
name(void) : string
/+ height(void) : float
/+ weight(void) : float
/+ sound(void) : string
+ sound(void) : string



4 bytes reinterpeted

(_height)

```
F1Car car(365, 4, "KUB");
Fish *nemo;

nemo = reinterpret_cast<Fish*> (&car);
cout << nemo->height() << endl;
```

Reinterpret F1Car as a Fish

TYPE CASTING

Const cast provides the mechanism of manipulating constness of an object:

- Constness can be set or removed
- Used mainly to pass a const object to a function that require non-const object as a parameter (or vice-versa)

```
void print (char * str) {
  cout << str << endl;
}

int main () {
  const char * s = "This is const string!";
  print ( const_cast<char*> (s) );
  return 0;
}
```

non-const param
(char *)

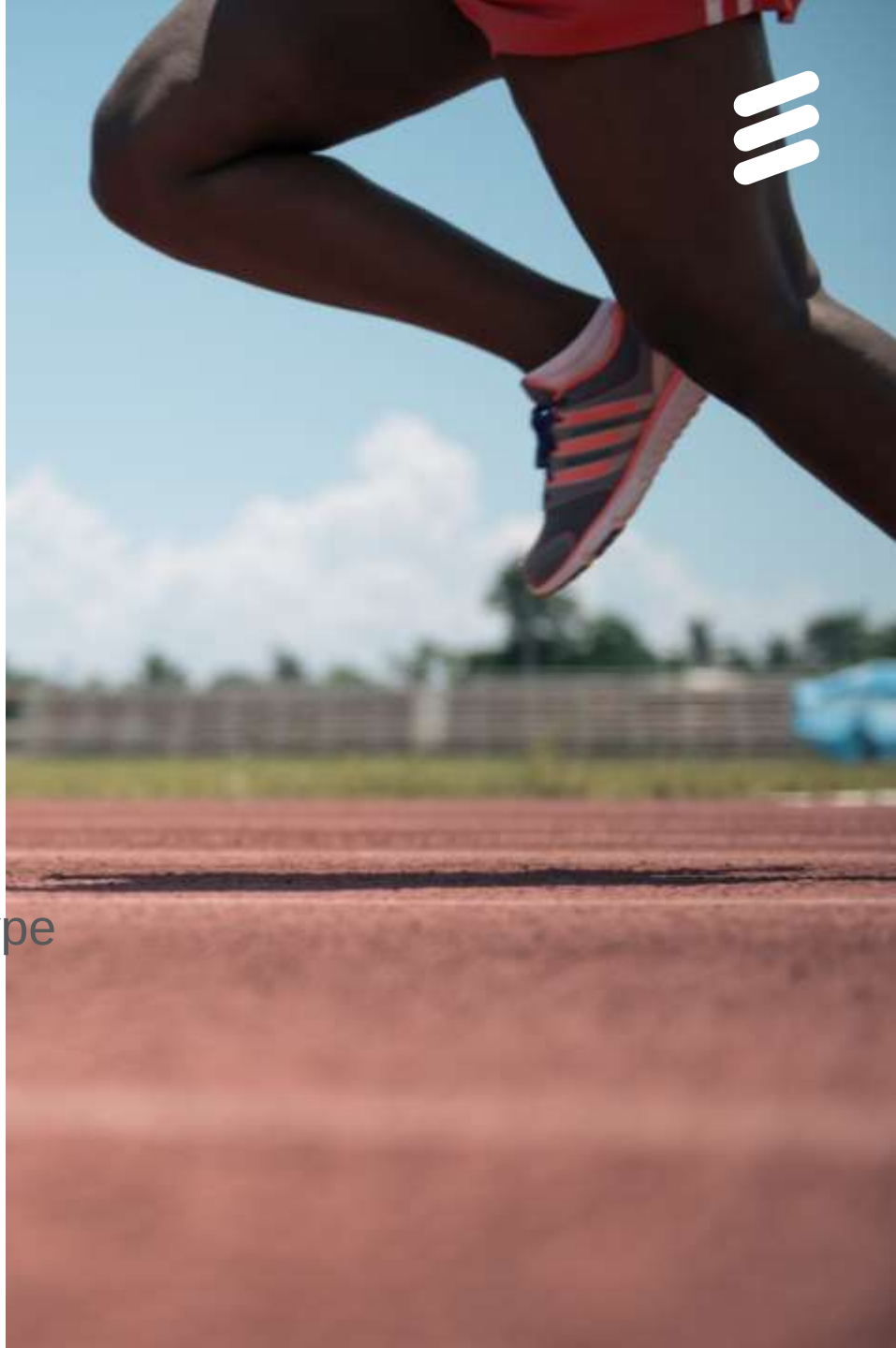
const variable
(const char *)

constness removed

TYPE CASTING

Summary:

- **Dynamic cast**
  - Pointers and references
  - Derived class to base class
- **Static cast**
  - Related classes pointers
  - Used for build types casting
  - Dangerous
- **Reinterpret cast**
  - Casting of memory area to any type
  - Very dangerous!
- **Const cast**
  - Constness added / removed



CHAPTER V: EXCEPTIONS

EXCEPTIONS

Exceptions provide a way to handle exceptional circumstances during runtime (runtime errors) using try-catch mechanism.

- Exception is thrown using `throw` keyword
- Can only be caught if thrown in `try` block
- Caught exceptions are handled in `catch` blocks
- There must be at least one `catch` block

```
try {
  <block of code>
  throw <exception object>;
}
catch(<exception object or ...>) {
  <exception handler block of code>
}
```

EXCEPTIONS

```
#include <iostream>

using namespace std;

int main()
{
  try {
    throw 100;
  }
  catch (int val) {
    cout << "Exception: " << val;
  }
  catch (...) {
    cout << "Unknown!" << endl;
  }

  return 0;
}
```

try block
(with throw)

exception block
(for `int` exception)

default exception block
(for all other types)

EXCEPTIONS

When declaring a function we can specify the types the function can throw as an exception:

```
double myFunction(double parameter) throw(int, double);
```

standard function declaration

thrown types

- If no throw keyword is used in function declaration – function can throw any data type as exception
- If no data types are defined in throw types part of definition function – functions exceptions can't be catched

`int fun(int val);`	All exceptions allowed
`int fun(int val) throw();`	No exceptions allowed

EXCEPTIONS

In order to declare own exception types:

- Include `<exception>` header
- Derive publicly from `exception` class
- Exception class has a virtual method "`what`" – providing exception description

```
class MyException: public exception
{
  const char* what() const throw()
  {
    return "My exception happened";
  }
};
```

EXCEPTIONS

```
#include <iostream>
#include <exception>

using namespace std;

int main ()
{
  try
  {
    MyException errorOccured;
    throw errorOccured;
  }
  catch (exception & e)
  {
    cout << e.what() << endl;
  }
  return 0;
}
```

`MyException` created

(and thrown)

`MyException` handled

(`My exception happened`)

EXCEPTIONS

Exceptions in practice**:**

- Project dependant
- Often ignored in Real-Time Systems
- Often `int` type is used for all exceptions just to indicate the error and its name (enumeration)

```
enum Exceptions
{
  timeout,
  userError,
  linkBroken,
  birdIsTheWord
}
```

EXCEPTIONS

Summary:

- **try-catch mechanism**
  - Only one try block
  - At least one catch block
  - Any data type can be used as exception
- **Exceptions in functions**
  - Function can throw any number of types of exceptions

CHAPTER VI: TEMPLATES

TEMPLATES

Templates are very powerful tool in C++ that allows:

- Single handling way for different data types
- Creation of unified code
- Great at creating containers

STL (Standard Template Library) is a collection of fully coded templates, for example:

- `vector`
- `bitset`
- `map`
- `...`

TEMPLATES

Template Functions allow creation of functions that can operate with generic types.

```
template <class T>
T GetMax (T a, T b)
{
  return (a > b ? a : b );
}

void main()
{
  int x = 10, y = 20;
  float a = 2.5, b = 5.5;

  cout << GetMax <int> (x, y) << endl;
  cout << GetMax <float> (a, b) << endl;
}
```

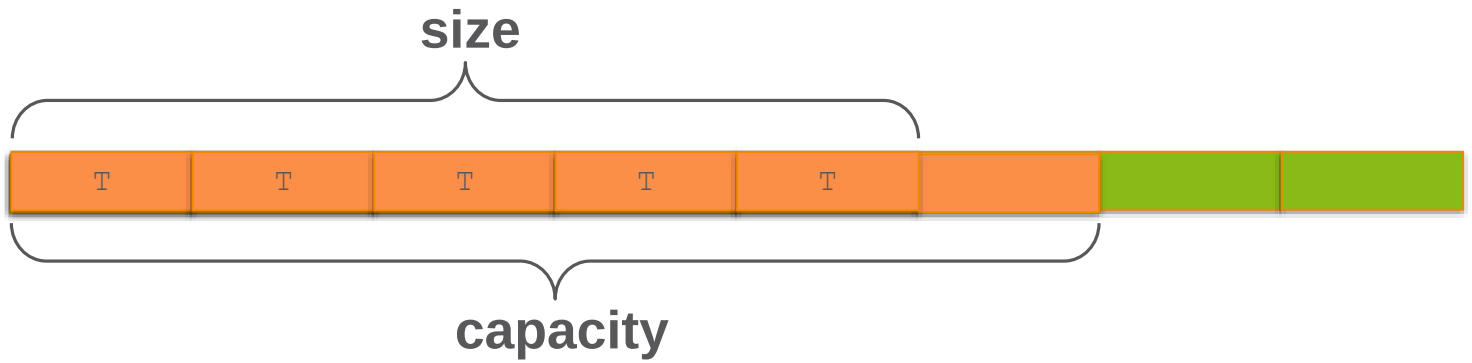
function template

TEMPLATES

Vector is a template similar to an array (organized memory block).

- Size dynamically changed
- Easy accessible

More information: http://www.cplusplus.com/reference/vector/vector/

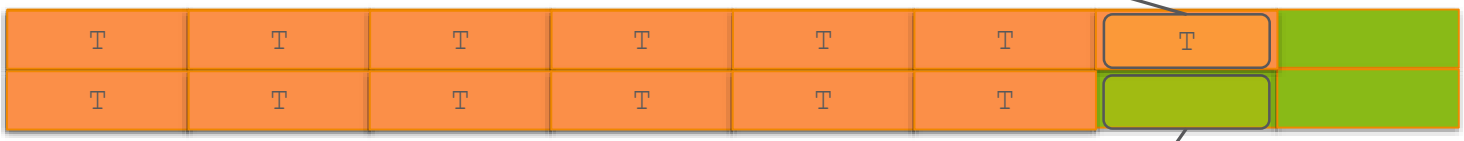


TEMPLATES

If there is not enough space in the vector to hold new object, the whole vector is moved in memory!

Capacity increased

(T added to the vector)



No capacity to add new T

(reallocation will occur)

TEMPLATES

```
#include <iostream>
#include <vector>

using namespace std;

void main () {
  vector <int> v;
  v.push_back(3);
  v.push_back(5);

  for (int i = 0; i < v.size(); ++i)
    cout << v[i] << endl;

  v.clear();
}
```

add value

(at the end)

TEMPLATES

Access to vector can be achieved using multiple methods.

```
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
{
  cout << *it << endl;
}
```

iterator for vector<int>

(similar to pointer)

algorithm

(for `for_each`)

```
void display(const int &i) { cout << i << endl; }

#include <algorithm>

for_each (const int &a, v, display);
```

`for_each`

(parameter, container, function)

TEMPLATES

Bitset is a template that holds bits:

- Good memory usage
- Simple in use

flags at 8 bits
(received or given)

number of bits

name
(constructor)

access
(i[th] bit)

```
#include <bitset>

int main()
{
  u8 flags = 92;

  bitset <8> options(flags);

  for (int i = 0; i < 8; ++i)
  {
    cout << options[i] << endl;
  }
  return 0;
}
```

TEMPLATES

Summary:

- **Functions**
  - Can operate on general types
- **STL**
  - Many containers
  - Easy to use
  - Well documented