C++ Programming
Lesson 3

TINPRO02-5B

HOGESCHOOL ROTTERDAM

PROGRAMMA

Recap

Classes, functions, attributes
Constructors & destructors
Function overloading
Constructor overloading,
Copy constructor

HOGESCHOOL ROTTERDAM

Recap Lesson 2

- Pointers
- References
- Address operators

Classes

```
class MyClass{
  public:
   //public members (accessible to all):
   //  functions
   //  types
   //  data
  private:
   //private members (used by members of this class
    only):
   //  functions
   //  types
   //  data
};
```

Classes

Class members are private by default:

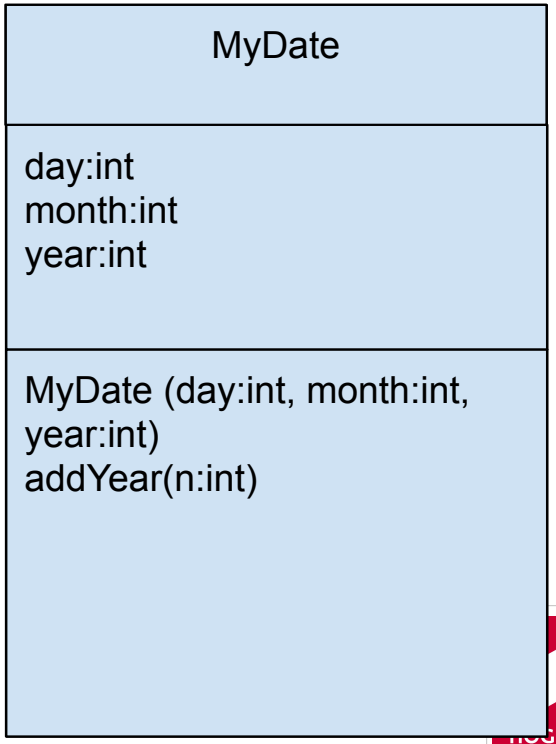
```
class MyClass{
  int mf(int);
…
};
```

means

```
class MyClass{
private:
  int mf(int);
…
};
```

Class declarations (in .h file)

```
class MyDate{
    private:
     int day,month,year;
    public:
     MyDate(int day, int month, int year); //constructor
     void addYear(int n); //add n years
 };
```



Class member functions (in .cpp file)

```
MyDate::MyDate(int d, int m, int y):
    day (d), month (m), year (y)   // Construction,
preferred
{
    /*
        day = d;  month = m; year = y;  // Assignment, less
desirable
    */
}

void MyDate::addYear(int n){
     year += n;
}
```


HOGESCHOOL ROTTERDAM

Function overloading

- If you have two or more function definitions for the same function name, that is called overloading.
- When you overload a function name, the function definitions must have different numbers of formal parameters or some formal parameters of different types.
- When there is a function call, the compiler uses the function definition whose number of formal parameters and types of formal parameters match the arguments in the function call.

Function overloading

```
void add(int a,int b){
  cout<<"Sum i, i: " <<a<< " + " <<b<< " = " << (a+b) << endl;
}
// Overload add for integer and float parameters
void add(int a,float b){
 cout<<"Sum i, f: " <<a<< " + " <<b<< " = " << (a+b) << endl;
}
// Overload add for two floating number parameters
void add(float a, float b){
 cout<<"Sum f, f: " <<a<< " + " <<b<< " = " << (a+b) << endl;
}

// The overloading is just for example, brings no benefits here
int main(){
    add(4,6);
    add(4,6.1);
    add(4.6,9.2);
 }
```

Class instances: stack and heap

Classes in C++ can be instantiated in 2 ways:

- On the stack
- On the heap

Class instances: stack and heap

Instances on the stack are automatically destructed when the function where they were created is finished. Class member functions are called using a dot, '.'.

```
MyClass anObject;
anObject.function ();
```

Class instances

Instances on the heap are not destructed automatically, so you must delete them manually to free memory.

This creates just a pointer to the object, and therefore the functions and attributes must be accessed by '->'.

```
MyClass* anObject = new MyClass();
anObject->function();
delete anObject;
```

Java does it this way, but Java cleans it automatically.

Note that anObject isn't an object but a pointer to it.

More about new and delete

The new operator should only be used if the data object should remain in memory until delete is called.

Otherwise if the new operator is not used, the object is automatically destroyed when it goes out of scope.

The objects using new are cleaned up manually while other objects are automatically cleaned when they go out of scope.

For more information:
https://www.tutorialspoint.com/when-to-use-new-operator-in-cplusplus-and-when-it-should-not-be-used

Constructors

- A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated.
- Used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used
- Constructors always have the same name as the class
- Constructors have no return type (not even void)

Constructors

- Constructors are only intended to be used for initialization.
- Do not try to call a constructor to re-initialize an existing object (it may compile, but the compiler will create a temporary object and then discard it).

Example

```
class Fraction{
public:
    int numerator;
    int denominator;
   //Constructor with no parameters
    Fraction(){
        numerator = 0;
        denominator = 1;
    }
};
int main(){
    Fraction f(); //Create an instance of Fraction on the stack
    cout << "num: " << f.numerator << ",  " << f.denominator << endl;
    return 0;
}
```

Constructor with parameters

```
class Fraction{
private:
    int numerator;
    int denominator;
 public:
    Fraction(int numerator){ // Constructor with one parameter
        this->numerator = numerator;  // Use this pointer to prevent confusion
        denominator = 1;
    }

    // Constructor with two parameters,
    Fraction(int numerator, int denominator){
        this->numerator = numerator;
        thisdenominator = denominator;
    }
};
```

Constructor with parameters

- Standard use:
  - `Fraction fiveThirds(5, 3);`
- From C++ 11
  - `Fraction fiveThirds {5, 3};`
  - `Fraction fiveThirds = {5, 3};`
- This will not work as expected
  - `Fraction fiveThirds = (5, 3);`

Constructor with parameters and default values

```
class Fraction{
private:
    int numerator;
    int denominator;
 public:
    // Constructor with two parameters, one parameter having a default value
    Fraction(int numerator, int denominator=1){
        this->numerator = numerator;
        this->denominator = denominator;
    }
};
 int main(){
     Fraction  f(5);// sets numerator = 5 and denominator = 1
     Fraction  f2(5,8);// sets numerator = 5 and denominator = 8

     return 0;
 }
```

It is possible to define default values

Default constructor

- A constructor that without arguments is called "**Default constructor**"
- If you define a class with no constructors at all, the the default constructor is automatically created (and does nothing)
- If your class definition includes one or more constructors with one or more arguments, but do not include a default constructor then there is NO default constructor

Calling default constructor

```
class DayOfYear{
    public:
        DayOfYear(int monthValue, int dayValue);
        DayOfYear(int monthValue);
        DayOfYear();//default constructor
    //...rest of code
};

int main(){
    DayOfYear date1(5, 8); //Calling constructor with 2 params
    DayOfYear date1(5); //Calling constructor with 1 param
    DayOfYear date1; //Calling default constructor

    return 0;
}
```

Pitfall with default constructor

- Do not use parenthesis for a call to a default constructor!!

```
DayOfYear date1(); //PROBLEM!!!
```

- For the compiler this is a function call to a function called `date1` that returns an object of type `DayOfYear`
- But you have to use parentheses when you explicitly invoke a constructor with no parameters:

```
date1 =  DayOfYear(); //Explicit call default constructor
```

Example with Initializer List (from C11)

```
class Fraction{
private:
    int numerator;
    int denominator;
 public:
    // Constructor with two parameters and initializer list
    Fraction(int numerator, int denominator): numerator(numerator),
        denominator (denominator)
    {} //Body can be empty now
};
  int main(){
      Fraction  f2(5,8);// sets m_numerator = 5 and m_denominator = 8
      return 0;
  }
```

Body CAN be empty, not necessarily

Compiler tips and tricks

- Some features like initializer list are available starting from c++ version 11 (C11)
- Compiler needs to know which C++ version it needs to compile
- `g++ -std=c++11 your_file.cpp -o main` — compiler version 4.7 or higher
- `g++ -std=c++14 your_file.cpp -o main` — compiler version 5.2 or higher
- `gcc -std=c++11 your_file.cpp -o main` — compiler version 4.7 or higher
- more info: https://gcc.gnu.org/projects/cxx-status.html

Constructor overloading

Just as it is possible to overload functions, it is possible to overload the constructor.

```
class DayOfYear{
    public:
        DayOfYear(int monthValue, int dayValue);
        DayOfYear(int monthValue);
        DayOfYear();//default constructor
    //...rest of code
};
```

Destructor

- It has the same name as the class but with the ~ symbol before the name
- It has no return type and no parameters
- Frees resources used by an object
- Function body is executed first then the members are destroyed
- Is used automatically whenever an object of its type is destroyed

Destructor

- A class can have only one destructor
- It is not possible to overload a destructor

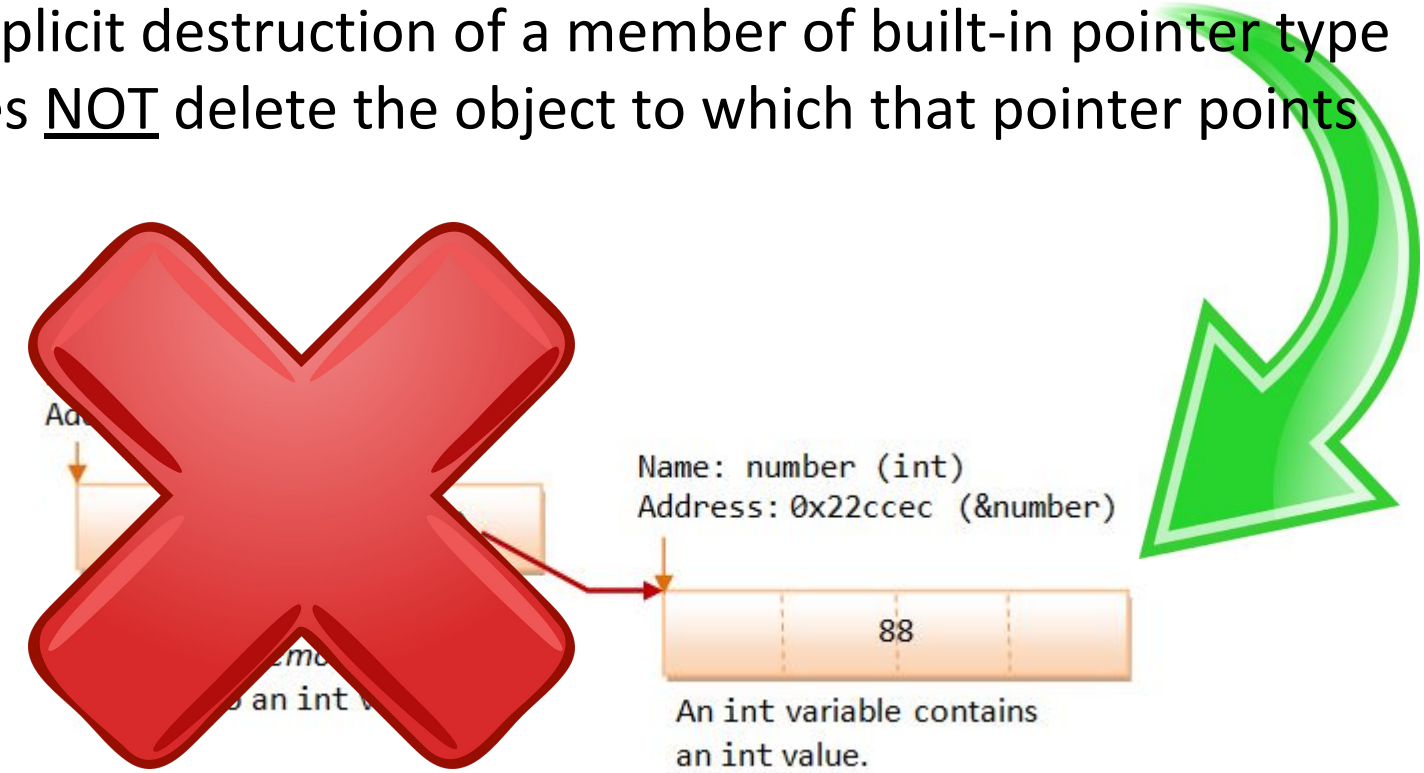
```
class Foo{
...
 public:
    virtual ~Foo() ; //Destructor
}
```

Destructor: tips and tricks

- Always make destructors virtual,
- If you don’t , it will cause problems with multiple inheritance.
- It is something which can be overlooked very easily.
- Only when working with very low amounts of memory it should be non-virtual.

Important!

The implicit destruction of a member of built-in pointer type does NOT delete the object to which that pointer points



Destructor: tips and tricks

- If the object dynamically allocates variables, the destructor must delete them
- Delete every dynamically allocated variables, when needed with many calls to **`delete`**
- The destructor may perform other clean-up detail
- It is something which can be overlooked very easily.
- Only when working with very low amounts of memory it should be non-virtual.

Example

```
class B {
    public:
        B() { cout << "Created `b`" << endl; }
        virtual ~B() { cout << "Deleted `b`" << endl; }
};
class A {
private:
    B* b;
    public:
        A() : b(new B()) { cout << "Created `a`" << endl; }
        virtual ~A() { delete b; cout << "Deleted `a`" << endl; }
};
int main() {
    A* a = new A();
    delete a;
}
```


HOGESCHOOL ROTTERDAM

Example

```
class String {
private:
    char *s;
    int size;
public:
    String(char *){
        size = strlen(c);
        s = new char[size+1];
        strcpy(s,c);
    }
    ~String(){
        delete []s;
    };
};
```

Example

```
//myclass.h

class MyClass
{
     private:
          int n;

     public:
          MyClass(int n);
          virtual ~MyClass();
};
```

```
//myclass.cpp

MyClass::MyClass(int n) :
 n(n)
{}

MyClass::~MyClass()
{
     cout << "Deleted " << n <<
endl;
}
```

Destroying references

Look at the code Destructors (previous page), what will happen if we do this:

```
int main()
{
    MyClass object1(1);          //Address 0x00C0FE
    MyClass* object2 = &object1;

    delete object2;
}
```

It will result in a 'segmentation fault', which means something is wrong with the memory. When `` `delete class2;` `` is called, the data at location '0x00C0FE' will be removed. At the end, the stack variable `` `class1` `` will be destructed automatically. Which will try to remove the data at '0x00C0FE' and that won't work since the data is already removed.

Encapsulation

Copy constructor

It is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

Copy Constructor example

```
//.h file
class Line {

   public:
      Line( int len );              // simple constructor
      Line( const Line &obj);  // copy constructor
      ~Line();                      // destructor
     int getLength( void );

   private:
      int *ptr;
};
```

Copy constructor

- It is a constructor that has one “call-by-reference” parameter of the same type as the class
- Normally the “call-by-reference” parameter is a constant (`const`)
- It is called automatically whenever a function returns a value of that class type
- Any class that uses pointers and the `new` operator should have a copy constructor

Copy Constructor example

See copyConstructorExample.cpp

Try it out!

- Write a class named Ball. Ball should have two private member variables with default values: color (“Black”) and radius (10.0). Ball should provide constructors to set only color, set only radius, set both, or set neither value. Also write a function to print out the color and radius of the ball.
- *Add constructors with default parameters. Use as few constructors as possible.*

Solution

Example of solution

BallClass2

Try it out!

- Create a class, name doesn't matter, it has a constructor, destructor and private field `int n`
- Constructor takes an integer as parameter and sets `n` to that value
- The destructor prints the value of `n`
- Create two instances of the class, one on the heap, the other on the stack
- Get them both destroyed

- Tips:
  - Make destructors virtual
  - Create 3 files

Solution

Example of solution

Destructors

Try it out!

- Take the code from the Ball exercise, and a new class named BallPit.
- BallPit contains a list of references to Ball instances, use an array for this.
- BallPit has 3 constructors, an empty one, one where you give an array and one where you give an amount of Balls and a color (defaults to black) and it will generate `n` balls with random sizes between 10 and 20.
- Also add methods to add a new Ball, remove a Ball, clear the whole pit and to print information about all the Balls in the pit
- Make sure the balls are removed from the memory using `delete`

Solution

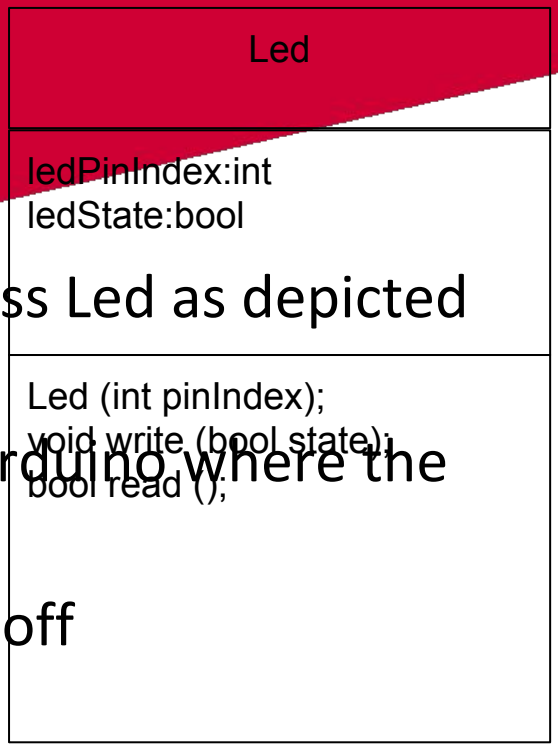
Example of solution

BallPit

Try it out!

- Create a class String, with at least the attribute size (int), and s (char *), where the string is stored
- Create a constructor with no parameters, a constructor with one parameter (char *), which initializes the string to the given parameter string
- *Create a constructor with 2 parameters, a given string and a given length*
- Create a suitable destructor
- *Create a copy constructor*
- Create the functions: copyString, getLength
- *Create extra functions for string manipulation*

Try it out!



In cantino, with your hardware, create a class Led as depicted in figure.

ledPinIndex: is the index of the pin on the arduino where the led is connected

pinState: indicates whether the led is on or off

Use the class to switch the led on and off

overtref jezelf