# C++ Programming I

## Basics of Object-Oriented Programming
## Destructor, Copy- & Move-Constructor

*C++ Programming*
*March 22, 2018*

Dr. P. Arnold
Bern University of Applied Sciences

# Agenda

- **Destructor**



# Agenda

- Destructor
- Copy Constructor



# Agenda

- **Destructor**
- **Copy Constructor**
- **Move Constructor**



# Agenda

- Destructor
- Copy Constructor
- Move Constructor
- Special Constructors



# Agenda

- Destructor
- Copy Constructor
- Move Constructor
- Special Constructors
- Exercise 05



# Destructor



# Constructor & Destructor

## Object-Cycle

- A **constructor** is a special initialization function (method) existing for every class
- The method is always called when an instance is created
- The constructor can define all values of the newly created instance
- An explicit initialization is no longer required!

1. Given by a construction plan any number of similar objects can be built
2. Within its lifetime each object fulfils its tasks, i.e. running through its states
3. When finished, the object is disposed automatically when out of scope
4. A **destructor** is a special function too! It's automatically invoked when a object is destroyed.

# Destructor

## Declaration of Destructor

```
1  // Declaration of a destructor inside the class, i.e. *.h file
2  class Human
3  {
4  public:
5      ~Human()
6      {
7          // destructor code here (.h)
8      }
9  };
10
11 // or
12
13 // Definition of a destructor outside the class's declaration
14 class Human
15 {
16 public:
17     ~Human(); // destructor declaration (.h)
18 };
19
20 // Implementation
21 Human::~Human()
22 {
23     // destructor code here (.cpp)
24 }
```

- The destructor takes the name of the class, but has a “~“ preceding it
- The role of destructor is the opposite to that of the constructor

# Destructor

## When and How to Use

- A destructor is always invoked when an object of a class is destroyed when it goes out of scope or is deleted via `delete`
- A destructor the ideal place to reset variables and release dynamically allocated memory and other resources like files etc, i.e. clean up

To understand the idea and use of the destructor we consider the example from the book (Listing 9.7).

- Although **not recommended** in C++ we use a dynamically allocated C-Style array of `chars` to hold a string
- To avoid allocating and deleting memory manually, we build our own class `MyString` to encapsulate the C-Style string

# Destructor

## When and How to Use

```
1  #include <iostream>
2  #include <string.h>
3  using namespace std;
4
5  class MyString
6  {
7  private:
8     char* m_buffer;
9
10 public:
11    MyString(const char* initString)   // constructor forces
          parameter
12    {
13       if(initString != nullptr) // check input!
14       {
15          m_buffer = new char [strlen(initString) + 1]; //
                allocate memory + terminator \0
16          strcpy(m_buffer, initString);  // copy to member
17       }
18       else
19          m_buffer = nullptr;
20    }
21
22    ~MyString() // destructor
23    {
24       cout << "Invoking destructor, clearing up" << endl;
25       if (m_buffer != nullptr) // check pointer!
26          delete [] m_buffer: // delete char array
27    }// --> continuing on next slide
```



Bern University
of Applied Sciences

# Destructor

## When and How to Use

```
1  // ...
2     int getLength()
3     {
4        return strlen(m_buffer);
5     }
6
7     const char* getString()
8     {
9             return m_buffer;
10    }
11 };
12
13 int main()
14 {
15    MyString sayHello("Hello from String Class");
16    cout << "String buffer in sayHello is " <<
           sayHello.getLength();
17    cout << " characters long" << endl;
18
19    cout << "Buffer contains: " << sayHello.getString() << endl;
20 }
21 // Output:
22 // String buffer in sayHello is 23 characters long
23 // Buffer contains: Hello from String Class
24 // Invoking destructor, clearing up
```

- A destructor cannot be overloaded! If you don't implement one, the compiler creates and invokes a dummy destructor

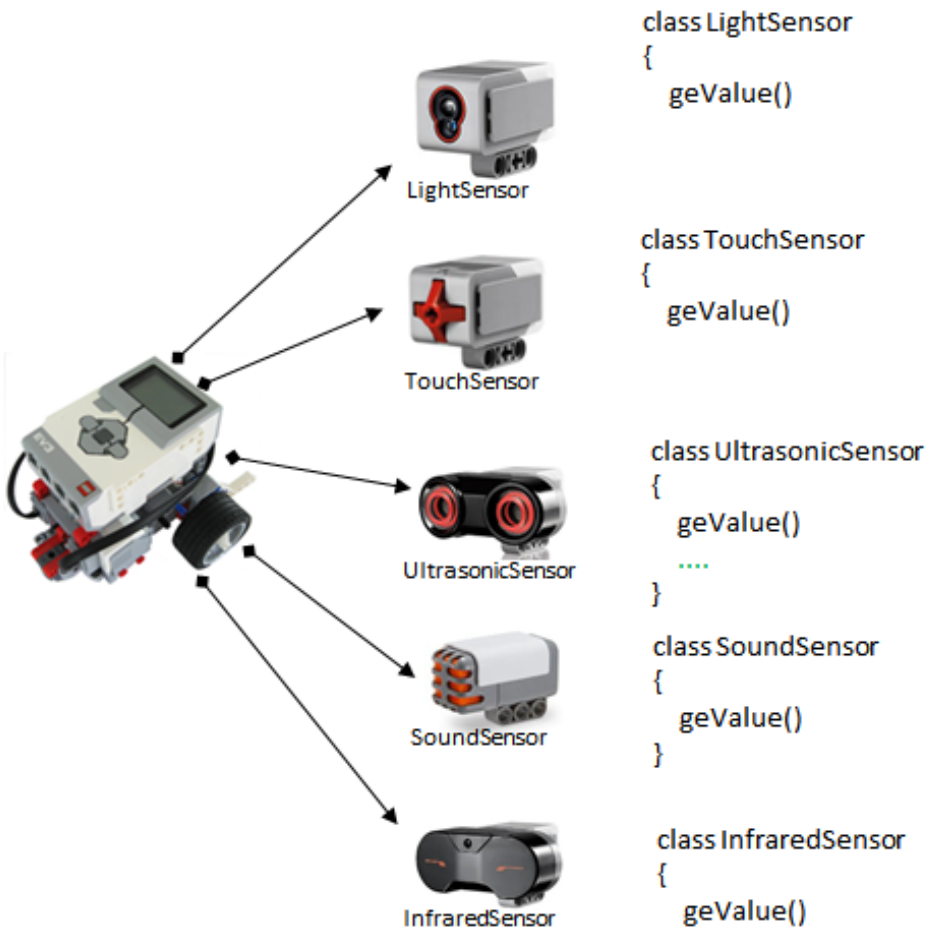
# Copy Constructor



# Copy Constructor

## Why we need a Copy Constructor

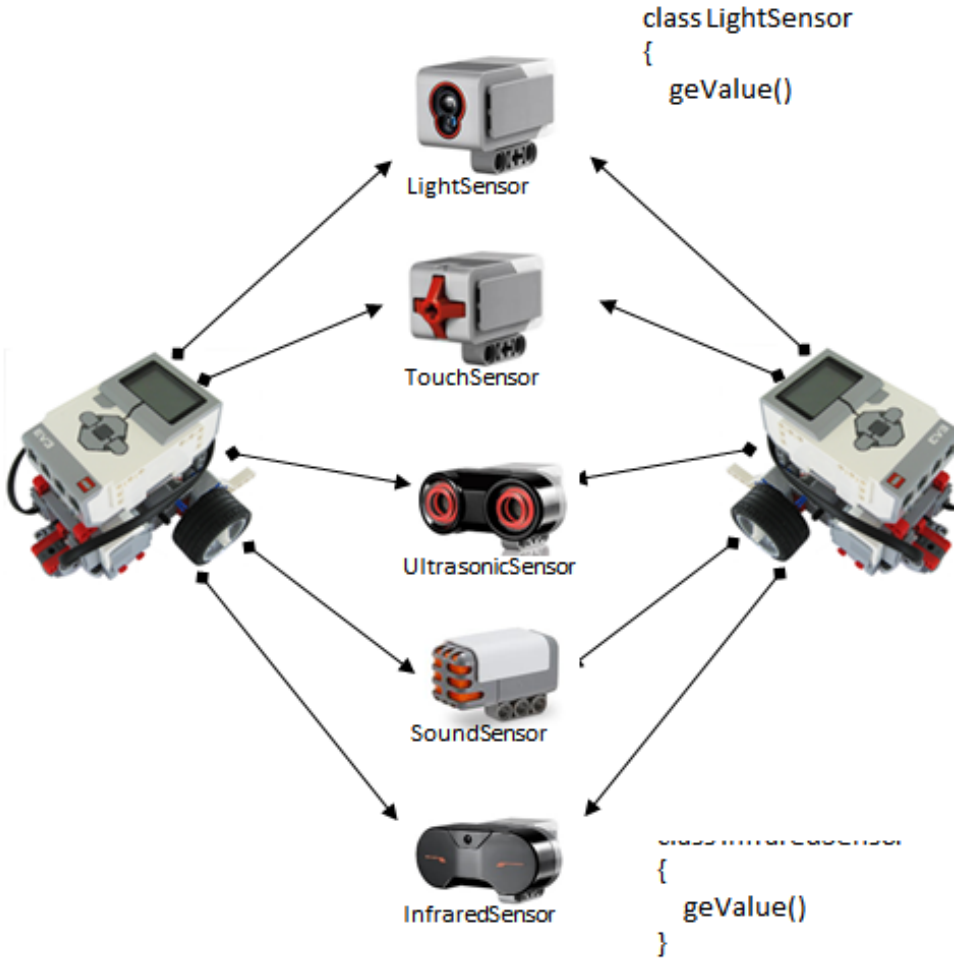
- Imagine you want to copy your class robot which holds **pointers** to the handles of each sensor



# Copy Constructor

## Why we need a Copy Constructor

- Doing a *shallow copy* of the handles (default copy constructor), both robots will use the same sensors!




- To guarantee a *deep copy* of the handles, a copy constructor must be provided

# Copy Constructor

## Why we need a Copy Constructor

- Consider the following example using our class `MyString`

```
1  #include <iostream>
2  #include "myString.h"
3  using namespace std;
4
5  void useMyString(MyString str)
6  {
7     cout << "String buffer in MyString is " << str.getLength();
8     cout << " characters long" << endl;
9
10    cout << "buffer contains: " << str.getString() << endl;
11    return;
12 }
13
14 int main()
15 {
16    MyString sayHello("Hello from String Class");
17    useMyString(sayHello);
18
19    return 0;
20 }
```

# Copy Constructor

## Why we need a Copy Constructor

- Consider the following example using our class `MyString`

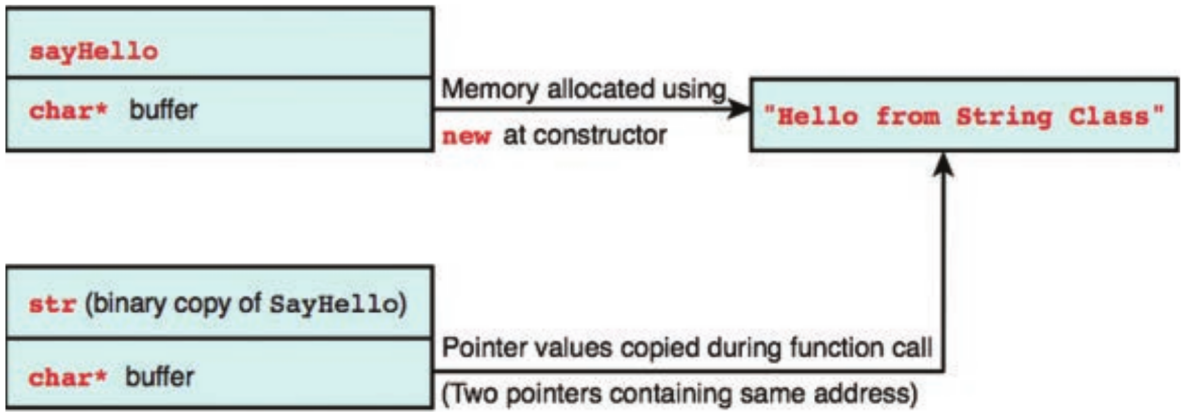
```
1  #include <iostream>
2  #include "myString.h"
3  using namespace std;
4
5  void useMyString(MyString str)
6  {
7     cout << "String buffer in MyString is " << str.getLength();
8     cout << " characters long" << endl;
9
10    cout << "buffer contains: " << str.getString() << endl;
11    return;
12 }
13
14 int main()
15 {
16    MyString sayHello("Hello from String Class");
17    useMyString(sayHello);
18
19    return 0;
20 }
```

```
1  // Output
2  String buffer in MyString is 23 characters long
3  buffer contains: Hello from String Class
4  Invoking destructor, clearing up
5  Invoking destructor, clearing up
6  // --> CRASH
```

# Copy Constructor

## Analysis

- Pass by value invokes copy! Since no copy-constructor was implemented, the compiler creates a default one




- Two objects of class `MyString` pointing to the same location in memory
- Delete is invoked twice - after function `useMyString` and `main`

**Warning!**

If we you use dynamic allocated memory in our classes, you have to implement a copy constructor to ensure **deep copy**

# Copy constructor

## Syntax

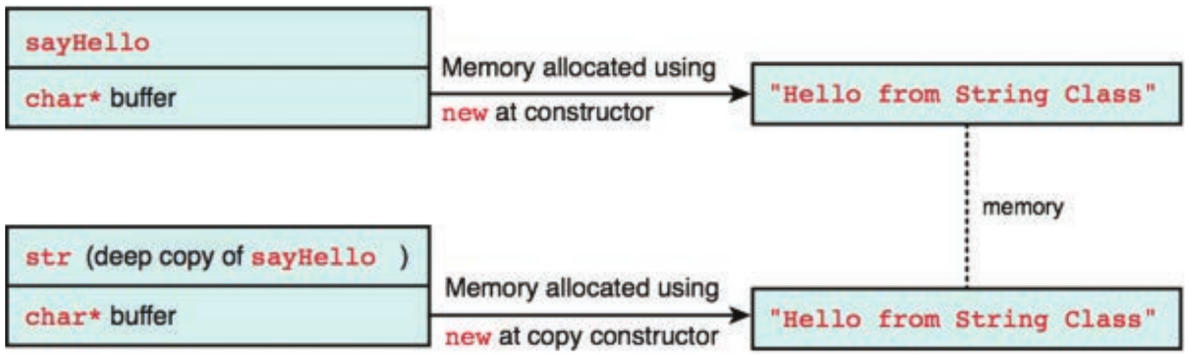
```
1  // Declaration syntax of a copy constructor for class MyString
2  class MyString
3  {
4      MyString(const MyString& copySource); // copy constructor
5  };
6
7  MyString::MyString(const MyString& copySource)
8  {
9      // Copy constructor implementation code
10 }
```

- A copy constructor takes an object of the same class by reference as a parameter
- Using **const** in the copy constructor declaration ensures that the copy constructor does not modify the source object being referred to
- The parameter in the copy constructor is **passed by reference** as a necessity. If this weren't a reference, the copy constructor would itself invoke a copy, thus invoking itself again and so on till the system runs out of memory

# Copy Constructor

## Ensuring Deep Copy for MyString

```
1  // Implementation of Copy-Constructor for MySting
2  MyString(const MyString& copySource) // Copy constructor
3  {
4      m_buffer = nullptr;
5      cout << "Copy constructor: copying from MyString" << endl;
6      if(copySource.buffer != nullptr)
7      {
8          // allocate own buffer
9          m_buffer = new char [strlen(copySource.m_buffer) + 1];
10         // deep copy from the source into local buffer
11         strcpy(m_buffer, copySource.m_buffer);
12         cout << "buffer points to: 0x" << hex;
13         cout << (unsigned int*)m_buffer << endl;
14     }
15 }
```



# Copy Constructor

## Ensuring Deep Copy for MyString

```
1  int main()
2  {
3      MyString sayHello("Hello from String Class");
4      UseMyString(sayHello);
5      return 0;
6  }
7
8  // Output:
9
10 // Default constructor: creating new MyString
11 // buffer points to: 0x01232D90
12 // Copy constructor: copying from MyString
13 // buffer points to: 0x01232DD8
14 // String buffer in MyString is 17 characters long
15 // buffer contains: Hello from String Class
16 // Invoking destructor, clearing up
17 // Invoking destructor, clearing up
```

- Two objects of class `MyString` pointing to different locations in memory
- The copy constructor has ensured deep copy in cases such as function calls:
- However, what if you tried copying via assignment:

```
MyString overwrite("who cares?  ");
overwrite = sayHello;
```

- This would still be a shallow copy! We have to supply a **copy assignment `operator=`** . Again the compiler has supplied a default for you that does a shallow copy

# Copy Constructor

## Summary

| DO | DON'T |
| --- | --- |
| **DO** always program a copy constructor and copy assignment operator when your class contains raw pointer members (`char*` and the like). | **DON'T** use raw pointers as class members unless absolutely unavoidable. |
| **DO** always program the copy constructor with a `const` reference source parameter. | |
| **DO** evaluate avoiding implicit conversions by using keyword `explicit` in declaring constructors. | |
| **DO** use string classes such as `std::string` and smart pointer classes as members instead of raw pointers as they implement copy constructors and save you the effort. | |

# Move Constructor



# Move Constructor

## Improve Performance

- There are cases where objects are subjected to copy steps automatically due to the nature of the language and its needs. For example return by value:

```
1  class MyString
2  {
3      // pick implementation from before
4  };
5
6  MyString copy(MyString& source) // function
7  {    // create copy
8      MyString copyForReturn(source.GetString());
9      // return by value invokes copy constructor!
10     return copyForReturn;
11 }
12
13 int main()
14 {
15     MyString sayHello ("Hello World of C++");
16     // 2x copy constructor invoked
17     MyString sayHelloAgain(copy(sayHello));
18     return 0;
19 }
```

- Purpose of move semantics is to avoid costly and unnecessary deep copying!
- Rvalue references allow to implement the so called *move semantics*

# Move Semantics

## Rvalue Reference

- Rvalue references are a feature of C++ added with the C++ 11 standard

```
1 // What is a rvalue reference?
2 int a = 5;  // a is a lvalue
3 int& b = a; // b is a lvalue reference
4 int&& c ... // c is a rvalue reference
```

- An lvalue is an expression that may appear on the left or on the right hand side of an assignment
- An rvalue is an expression that can only appear on the right hand side of an assignment

**Note!**

`&&X` is not a reference to a reference of `X`!
`&&X` is a rvalue reference to X.

# Move Semantics

## Rvalue Reference - Basic Concept

- With rvalue references functions and constructors can be overloaded

```
1  // function overloading
2  printInt(int& i){cout << "lvalue reference: " <<i<< endl;}
3  printInt(int&& i){cout << "rvalue reference: " <<i<< endl;}
4
5  int main()
6  {
7      // What is a rvalue reference?
8      int a = 5;    // a is a lvalue
9      printInt(a); // Call printInt(int& i)
10     printInt(6); // Call printInt(int&& i)
11
12     return 0;
13 }
```

- In this example there is no performance gain, since the argument `i` is very small!
- Let's return to the more realistic scenario of the `MyString` class, where `MyString` can hold a very large char array

# Move Constructor

## Improve Performance

- To avoid not necessary copies a `move`-constructor can be implemented
- For the move constructor rvalue references are used

```
1  // move constructor – change ownership
2  MyString(MyString&& moveSource) // arg is rvalue reference
3  {
4      if(moveSource.m_buffer != nullptr)
5      {
6          m_buffer = moveSource.m_buffer; // take ownership
7          moveSource.m_buffer = nullptr; // set nullptr!
8      }
9  }
10
11 // or shorter init list
12 MyString(MyString&& moveSource)
13     : m_buffer(moveSource.m_buffer) // take ownership
14 {
15     moveSource.m_buffer = nullptr; // set nullptr!
16 }
17
18 // Output
19 MyString sayHelloAgain(copy(sayHello));
20 // invokes 1x copy & 1x move constructors
21 // Note: copy(sayHello) is rvalue reference
```

# Special Constructors



# More Constructors

## Special Cases

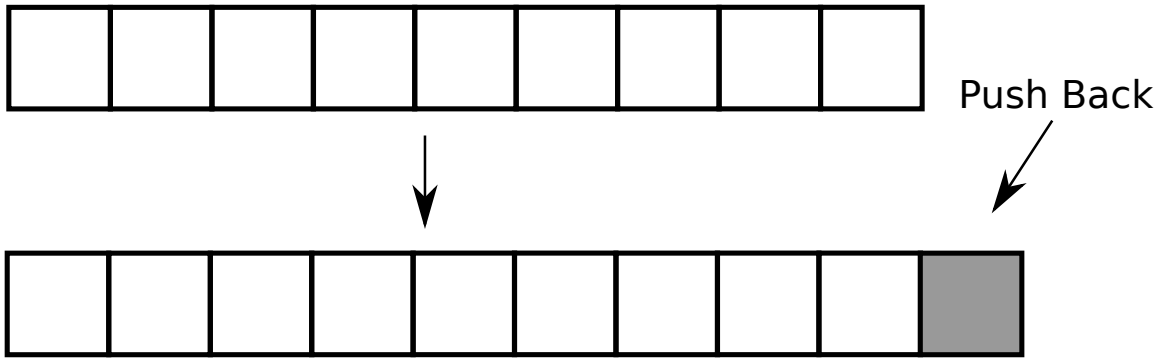
Please read Lesson 9 of the book to repeat the most important details and to learn more about:

- Class that does not permit copying
- Singleton class that permits a single instance
- Class that prohibits instantiation on the Stack
- Constructors that convert types

# Exercise 05 - Hint

## Vector `push_back()`

- How to implement the `push_back` functionality? How to resize an array?




1. Create new array of size + 1
2. Copy elements
3. Delete old array

# Thank You

## Questions

???