Constructors, Destructors, Pointers and References

Mikael Svahnberg*

2023-09-22

Contents

*Mikael.Svahnberg@bth.se

1 Introduction

- Constructors and Destructors
- Copy Constructors
- Pointers and References
- Identity and Equality
- Operator Overloading
- Constants

Today, we start in C++.

2 Constructor

- Setup of a new object
  - Initialise all attributes
  - Copy any data given to the object to take ownership of it
    - We do not know what the creator will do with the data once this object is created
    - It can go out of scope at the end of a function
    - It can be deleted
- Looks like a normal method
  - Same name as class (case-sensitive)

  - No return value: the return of a constructor is the created object
  - *Should* have no side-effects: just create the object

- Can have many constructors
  - Default constructor takes no parameters `className::className()`
  - Can be used to convert to the class type
    * e.g. `className::className(int)` will be used to "upgrade" an `int` to className.

- Constructors can be `private`
  - can only be called from within the class and from `friends` (which we will not cover)

3 Example of "normal" Constructors

3.1 Header

```
#pragma once     // A more modern way (but not standard) instead of the #ifndef below
#ifndef WORD_HH
#define WORD_HH
#include <string>
using namespace std;

class Word {
public:
  Word();
  Word(string theWord, int theOccurrences);
  Word(string theWord);
  Word(char* theWord);
  Word(int theOccurrences);

  void print(void);
private:
  string myWord;
  int myOccurrences;
};

#endif
```

3.2 Implementation

```
#include <string>
#include <iostream>
#include "word.hh"

using namespace std;

Word::Word() : Word("",0) {};
```

```
Word::Word(string theWord) : Word(theWord,0) {};
Word::Word(char* theWord) : Word(string(theWord)) {};

Word::Word(string theWord, int theOccurrences) : myWord(theWord),myOccurrences(theOccurren
  this->print(); // print() is a side-effect, so we should avoid this.
}

Word::Word(int theOccurrences) : Word("", theOccurrences) {};

void Word::print(void) {
  cout << "Word: " << myWord << " occurrences: " << myOccurrences << endl;
}
```

4 Creating Objects

```
#include <iostream>
#include "word.hh"

using namespace std;

void testWord(void) {
  cout << "Testing Word..." << endl;
  Word first = Word("first",1);
  Word* second = new Word();
  Word* third = new Word("third", 1);

  // This works, but is not preferred
  Word fourth("fourth", 1);


  // Make use of the other constructors to "upgrade" a string and a char*
  Word fifth = "fifth";
  Word sixth = string("sixth");
  Word seventh = 10;  // Upgrade an int... to Word(10), which calls Word("",10)



  // Careful! This does not do what you expect (it actually declares a new function)
  Word ninety_nine();
  Word ninety_eight; // This is what you wanted to do
  Word ninety_seven = Word(); // or this, which is the same thing
  cout << "Done Testing Word..." << endl;
}
```

5 Copy Constructor

```
#include "word.hh"
```

```
int main(void) {
  Word aWord = Word("something", 1);
  Word anotherWord = aWord;  // This will use a built-in constructor
}
```

- This is usually *not* what we want!
- The built-in constructor copies every member attribute straight off.
  - ok for built-in data types
  - ok for objects that have a copy constructor (but how do you know?)
  - *not ok for pointers* – only the reference is copied, not the contents.
  - not ok for objects without copy constructors
- We want to define our own *copy constructor*:

```
Word::Word(const Word& original) {
  // We take one and only one parameter of the same type as ourselves
  // it is declared as const since we must not modify it
  // it is a reference (Word&) since we do not want to accidentally copy it.

  myWord = original.myWord; // Create a clone of the string.
  myOccurrences = original.myOccurrences;
}
```

6 Example using the Copy Constructor

- Make use of both old-school c-strings `char*` and `string`
  - c-strings are arrays of characters, null-terminated (the last character is `'\0'`)
  - for the colours, we need our own way to terminate:
    - We could pass around the number of colours
    - We use an empty string `""` to signal that this is the last colour.
- `while(){}` and `do {} while ()`
- Using built-in arrays, we break the design principle *High Cohesion*
  - A Cat has to contain methods to manage an array of colours: `listColours()` and `copyColours()` .

6.1 Header

```
#ifndef CAT_HH
#define CAT_HH
#include <string>
#include <cstring>
using namespace std;
```

```
class Cat {
public:
  Cat();
  Cat(const char* theName, string* theColours);
  Cat(const Cat& original);

  void setName(const char* theName);

  string toString(void) const;
  string listColours(void) const;
private:
  void copyColours(const string* theColours);

  string* myColours = 0; // Give attributes a default value.
  char* myName = 0;      // Otherwise, they get whatever junk value that happened to be in
};
#endif
```

6.2 Implementation

```
#include <string>
#include <cstring>
#include <iostream>
#include "cat.hh"
using namespace std;

Cat::Cat() {
  string colours[] = {"grey", ""};
  Cat("youthere", colours);
}

Cat::Cat(const char* theName, string* theColours) {
  setName(theName);
  copyColours(theColours);
  cout << "Created " << toString() << endl;
}

Cat::Cat(const Cat& original) { // Could break up original and pass on to the other constr
  setName(original.myName);
  copyColours(original.myColours);
  cout << "Copy of " << original.toString() << endl;
}

void Cat::setName(const char* theName) {
  char* oldName = myName;
  myName = new char[strlen(theName+1)];
  strcpy(myName, theName);

  if (0 != oldName) {
    cout << oldName << " is now known as " << myName << endl;
```

```
  }

  // need to free oldName, otherwise you have a memory leak. There is no garbage collector
  delete [] oldName;
}

string Cat::toString(void) const {
  return string("a cat by the name ") + myName + " with the colours " + listColours();
}

string Cat::listColours(void) const {
  string out;
  int pos = 0;

  while ("" != myColours[pos]) {
    out += myColours[pos] + ", ";
    pos++;
  }

  return out;
}

void Cat::copyColours(const string* theColours) {
  int length=0;
  while ("" != theColours[length++]) {}
  myColours = new string[length];

  int pos=0;
  do {
    myColours[pos] = theColours[pos];
  } while ("" != theColours[pos++]); // Sometimes we want to do the loop check at the end:
}
```

6.3 Usage

```
#include <iostream>
using namespace std;

#include "cat.hh"

void testCat(void) {
  cout << "Testing cat" << endl;
  string colours[] = {"Red", "Orange", "White", ""};

  Cat* tabby = new Cat("Tabby", colours);
  cout << "Printing " << tabby->toString() << endl;

  Cat* copycat = tabby; // This just copies the pointer, no copy constructor is called.
```

```
  Cat bob = Cat("Bob", colours);
  Cat steve = bob;
  cout << "Printing Steve " << steve.toString() << endl;
  steve.setName("Steve");
  cout << "Printing Steve and Bob " << steve.toString() << bob.toString() << endl;

  steve = *tabby; // Get the object referred to by the pointer 'tabby', assign it to steve
                  // We would hope that this had invoked a copy constructor.
                  // It doesn't, since we are not *creating* objects, merely re-assigning
  steve = Cat(*tabby); // Creates a new Cat object using the copy constructor, and assigns

  colours[0] = "grey"; // We copy each colour, not just the array reference, so this will
  cout << "Printing Steve " << steve.toString() << endl;


  cout << "Done Testing cat" << endl;
}
```

7 Stack and Heap

- Normal variables are created on the *Stack*
  - the stack grows and shrinks for every method call and return
  - variables are created when declared, and removed when they go out of scope (e.g. `return` or end of the block)
- Use `new` to allocate objects on the *Heap* if you want to create an object that should live longer than the current scope

```
void someFunction(int copiedParameter, Cat& borrowedParameter, Cat* copiedPointer) {
  string colours[] = {"black", ""};

  copiedParameter = 10; // Will only be relevant in this function
  borrowedParameter.setName("Neo"); // Will change the original object
  copiedPointer->setName("Leo"); // Will also change the original object

  copiedPointer = new Cat("Louie", colours); // Will create a new Cat,
  // and *locally* change copiedPointer. Upon return, copiedPointer goes out of scope, and
}


Cat* badFunction() {
  string colours[] = {"black", ""};
  Cat garry = Cat("Garry", colours); // New cat created on the stack

  return &garry; // BAD! Yes, I can return the address to any object, but
                 // a few method calls later, the stack will be overwritten and Garry will
}

Cat* okFunction() {
```

```
  Cat* edward = new Cat("Edward", colours);
  return edward; // Ok, Edward is created on the heap, the pointer is returned
}
```

8 Delete

- In C++, you manage memory yourself, using `new` to allocate memory on the *heap*
- When a variable goes out of scope or you reassign a pointer, this memory is lost
  - Known as a *Memory Leak*
  - As a program executes, you risk running out of memory.
- Use `delete` to free the memory so that it can be used again.
- Make it a habit that for every `new` there is a `delete`.

```
int main(void) {
  int* number = new int(12);
  // 1. Allocate space for a pointer on the stack, and set 'number' to use this space
  // 2. Allocate space for an integer on the heap, set the value of this space to 12, retu
  // 3. Assign the pointer 'number' to the address of the created int.

  cout << *number << endl; // Access the value in the address pointed to by the variable '
  delete number; // Free the memory for the integer from the heap, and make it available t
}
```

9 Array Delete

```
#include <iostream>
using namespace std;

#include "cat.hh"
#include "dog.hh"

void testCatarray() {
  cout << "Testing array delete" << endl;

  Dog* d = new Dog();
  delete d;

  Dog* myDogs = new Dog[5];
  // delete myDogs; // This only tries to delete the first of the five dog objects.
                    // NOT any memory allocated by these objects.

  Dog* myDogs2 = new Dog[5];
  delete [] myDogs2;
```

```
  // This first calls delete on every object in the array
  // Then it deletes the array itself.

  cout << "Done Testing array delete" << endl;
}
```

10 Destructor

- Every well behaved class should have a *Destructor*
  - Close any open files, open network connections, etc.
  - Delete any attributes that were created on the heap.
  - In general, clean up after itself.
- `className::~className() {}`
- Often declared `virtual` to allow sub-classes' destructors to also run.

```
class Dog {
public:
  Dog() { cout << "Woof, created" << endl; }
  virtual ~Dog() { cout << "Woooof, deleted" << endl; }
};
```

11 Operator Overloading

- In Java, we only use method calls to operate on objects.
- This is a conscious design decision because it makes the code clearer.
- However,
  - Each class may invent their own names to do the same thing
  - Classes cannot be used seamlessly as if they were built in
- In C++, classes are seen as extensions of the language
- We should be able to use objects in the same way as we do with built-in data types.
- We can thus overload operators, e.g. `+-/*`, but also equal `==`, assignment `=`, and all others.

+	-	*	/	%	^	&	
++	–	«	»	==	!=	&&	
+=	-=	*=	/=	%=	^=	&=	
~	!	,	=	<	>	<=	>=
[]	()	«=	»=	->	->*	new	delete

Also: |, ||, |=

- In fact, all but `::`, `.*` , `.` , and `?:` can be overloaded.

12 Example – Java vs C++ with Operator Overloading

- Which version is actually easier to read and understand?

```
FancyBuffer inBuf;
while (readString(System.in, inBuf) ) {
  if ( inBuf.isEmpty() ) return;
  if ( inBuf.equals("done") ) return;

  switch ( inBuf.charAt(0) ) { /* ... */ }

  FancyBuffer copy;
  copy = inBuf.clone();

  System.out.println(inBuf.toString());
}
```

```
FancyBuffer inBuf;

while ( cin >> inBuf ) {
  if ( !inBuf ) return;
  if ( inBuf == "done") return;

  switch ( inBuf[0] ) { /* ... */ }

  FancyBuffer copy;
  copy = inBuf;

  cout << "Buffer as a string: " << inBuf << endl;
}
```

13 Example of some overloaded operators

13.1 Header

```
#ifndef RAT_HH
#define RAT_HH

#include <iostream>
#include <string>
using namespace std;

enum Gender {female=1, male=2, other=3, unspecified=99};

class Rat {
public:
  // Good practice to always include default constructor, copy constructor, and destructor
  Rat();
  Rat(Rat& original);
```

```
  virtual ~Rat();  // virtual so that sub-classes' destructors will also be called

  Rat(string name, Gender gender);

  // Some overloaded operators
  // The return type can often be anything we want but some make more sense than others.
  Rat& operator=(const Rat& other);
  bool operator==(const Rat& other) const;
  Rat* operator*(const Rat& other) const;

  friend ostream& operator<<(ostream& os, const Rat& theRat);
  // Yet another weird C++ quirk: friends can access the internals of a class
  // In this case, it is needed to get the << operator to work since the first
  // parameter is of another type than the class itself.
private:
  string name;
  Gender gender;
};

#endif
```

13.2 Usage

```
#include "rat.hh"

void testRat(void) {
  cout << "----------Assignment----------" << endl;
  Rat r("Manny", Gender::male);
  Rat a,b;
  cout << a << endl; // Tests operator<<() as well;
  a=r;
  b=a=r; // This is the reason why we return a reference to *this in operator=;

  cout << a << " and " << r << endl;

  cout << "----------Equals----------" << endl;
  Rat imitator("Manny", Gender::male);
  if (imitator == r) {
    cout << imitator << " and " << r << " are the same" << endl;
  } else {
    cout << imitator << " and " << r << " are NOT the same" << endl;
  }

  Rat imitator2("Mary", Gender::other);
  if (imitator2 == r) {
    cout << imitator2 << " and " << r << " are the same" << endl;
  } else {
    cout << imitator2 << " and " << r << " are NOT the same" << endl;
  }
```

```
  cout << "----------Multiplication----------" << endl;
  Rat fr("Mimmi", Gender::female);
  Rat* litter = r*fr;
  if (0 != litter) { cout << "First rat in mischief is " << litter[0] << endl; }
}
```

13.3 Impementation

```
#include <iostream>
#include <string>
using namespace std;

#include "rat.hh"

Rat::Rat() : Rat("no-name", Gender::unspecified) {
};

Rat::Rat(Rat& original) {
  this->name = original.name;
  this->gender = original.gender;
}

Rat::~Rat() {
}

Rat::Rat(string name, Gender gender) {
  this->name = name;
  this->gender = gender;
}

Rat& Rat::operator=(const Rat& other) {
  this->name = other.name;
  this->gender = other.gender;
  cout << "Assignment: " << *this << endl;
  return *this;
}

bool Rat::operator==(const Rat& other) const {
  cout << "Equals?: " << *this << " vs " << other << endl;
  return ((this->name == other.name) &&
          (this->gender == other.gender));
}

Rat* Rat::operator*(const Rat& other) const {
  if ((this->gender == male && other.gender == female) ||
      (this->gender == female && other.gender == male) ||
      (this->gender == unspecified || other.gender == unspecified)) {
    Rat* mischief = new Rat[5];
    return mischief;
  } else {
```

```
    return 0;
  }
}

ostream& operator<<(ostream& os, const Rat& theRat) {
  string gender;
  switch (theRat.gender) { // friends can access private attributes
  case Gender::female:
    gender = "female";
    break;
  case Gender::male:
    gender = "male";
    break;
  case Gender::other:
    gender = "other";
    break;
  case Gender::unspecified:
  default:
    gender = "unspecified";
  }
  return os << "Rat of " << gender << " gender called " << theRat.name;
}
```

14 Some (repeated) Points about Pointers

```
string colours[] = {"gray", ""};
Cat* cicero; // create a pointer. This allocates space for an address (64 bit?) on the sta

cicero = new Cat("Cicero", colours); // Create space for a cat on the heap. put the addres
// A cat is a char pointer and a string pointer. Nothing more.
// The cat's constructor, in turn, allocates space on the heap for the name (char*) and th

Cat ref = *cicero; // Dereference the pointer so that it act as the object itself.
cicero->meow(); // method call on an object referenced by a pointer.
ref.meow();     // method call on an object referenced by a "normal" variable

Cat calligula("Calligula", colours);
Cat* emperor = &calligula;  // Get the address of the variable.
Cat* writer = &cicero; // BAD: writer now holds the address to the *pointer* cicero, not t
```

15 Function/Method Parameters

```
// Pass-by-value
void someFunction(Cat theCat); // Declares a function (to be implemented later). Copies th
someFunction(calligula); // This creates a copy of calligula inside someFunction()

void otherFunction(Cat* theCat); // copies the *pointer*, but refers to the same object
```

```
otherFunction(cicero); // Will access the same Cicero cat inside the function as a pointer

// Pass-by-reference
void yetOneFunction(Cat& theCat); // "borrows" a reference to the same cat
yetOneFunction(calligula); // Will access the same Calligula cat inside the function as a
```

16 null pointers and null objects

- A pointer to the address `0` is called a *null* pointer
- A null pointer never points to a valid object
- Used as a default value:
  - null → not yet set
  - null → unable to complete
- Can lead to "defensive" programming
  - Have to check for null everywhere.
- Often better to use a *null object*
  - A proper object
  - created as normal
  - can be passed around (returned, used as parameters, etc.) as normal
  - behaves as normal; methods can be called and work as expected
  - contain real data *but* which does not mean anything

17 Arrays

- An array is a pointer to a *continuous segment of memory*
  - values and objects (of the same type) are lined up nose-to-tail in this memory
  - objects can be of any size
  - values can be built-in data types. *including pointers*
    * pointers are important since this allows an array to contain different types of objects (within the same inheritance hierarchy)
    * pointers can also refer to another array; This is how we create a two-dimensional array
- Remember to save the pointer to the first object; if you loose it, you won't find the start again
  - Use a second pointer to iterate over the objects

```
Cat* aCatArray = new Cat[5]; // An array of five cats;
Cat** aCatMatrix = new Cat*[5]; // An array of five cat pointers
for (int i = 0; i < 5; i++) {
  aCatMatrix[i] = new Cat[6]; // We now have a 5x6 matrix of cats.
}

aCatArray[0] // Access the first cat
aCatMatrix[0][0] // Access the top-left cat.

Cat* catIterator = aCatArray; // catIterator points to the first Cat
catIterator++; // Now the second cat
```

18 Constructors in Java

- Constructors work similarly to C++

```
public class Cat {
  public String name;
  public ArrayList<String> colours;

  Cat() { // Default constructor
    colours = new ArrayList<>();
    colours.add("Brown");
    name = "youthere";
    System.out.println("Default Constructor " + name);
  }

  Cat(String theName, ArrayList<String> theColours) {
    name = theName;
    colours = theColours;
    System.out.println("Constructor " + name);
  }

  Cat(Cat original) { // Copy constructor
    name = original.name; // Strings are special, so this one behaves as expected
    colours = original.colours; // But be careful with other object references. See in the
    System.out.println("Copy Constructor " + name);
  }

  public String toString() {
    return "a cat with the name " + name + " and the colours " +colours.toString();
  }

  public static void doSomething(Cat theCat) {
    System.out.println("Doing something with " + theCat.toString());
  }

  public static void main(String [] args) {
    Cat stray = new Cat();
```

```
    System.out.println("Stray: " + stray.toString());

    //Cat mongrel = stray; // Create a new reference 'mongrel', and points it at the same
    Cat mongrel = new Cat(stray); // Invoke the copy constructor

    mongrel.name="Manny";
    mongrel.colours.add("White"); // This is where the "Just copy the reference" will bite
    System.out.println("Mongrel: " + mongrel.toString());
    System.out.println("Stray, as it is now: " + stray.toString());

    doSomething(mongrel); // Will only pass the reference, less risk of accidentally copyi
  }
}
```

19 Destructor in Java

- `public void finalize() {}`
- `finalize()` is being removed from Java (*Deprecated*), avoid using it.
- Is run when the garbage collector decides to clean up the object
  - ... Which may be too late
- Can not rely on this ever being run. Or in any partcular order.
- Use to make sure that open files and ports are closed. Do not rely on this.
- Garbage Collector runs in its own thread.
- We can force the gc to run, but normally we do not need to.

```
public class Dog {
  public String name;

  public Dog() { this("Goodboy"); }
  public Dog(String theName) {
    name = theName;
    System.out.println("Create " + name);
  }

  public Dog(Dog original) { // Copy Constructor
    name = original.name;
    System.out.println("Copy " + name);
  }

  public void finalize() {
    System.out.println("Closing down and clenaing up " + name);
  }

  public static void main(String [] args) {
    Dog danny = new Dog();
```

```
    Dog clony = new Dog(danny); // This invokes the copy constructor
    clony.name = "ClonyBoy";
    System.out.println("Danny: " + danny.name +"\nClony: " + clony.name);

    Dog clonyclony = clony;

    danny = null; // No remaining references to danny, will be garbage collected
    clony = null; // ClonyBoy still has one reference.
    System.gc();
    System.out.println("Just ran the Garbage Collector");
  }
}
```

20 Pointers and Java: Pass by Value

- Anything created by `new` is an object
- All objects are accessed by a *reference*
- References mostly work as c++ pointers

```
Cat mierda = new Cat("Mierda");
Cat caca;

caca = mierda; // Copy the reference to the same object

doSomething(mierda); // Pass along a reference to the same object
```

- In method calls, Java *always use pass by value*
  - as expected with built-in data types
  - when the parameter is a reference to an object, the value passed is the *reference*, not the object.
  - Pass by reference is not possible. Use return to return modified element instead.

```
public static void doSomething(String someString) {
  someString = "not hello"; // Only changes the reference locally
}

public static String doChange(String someString) {
  return "modified " + someString;
}

public static void main(String [] args) {
  String s = "hello";
  doSomething(s);
  System.out.println(s);
  System.out.println(doChange(s));
}
```

21 Identity and Equality

- Common to C++ and Java
- **Identity** is not the same as **Equality**
- *Shallow* vs *Deep equal*

```
+----------------+      +-------------------+
| Type: Cat      |      | Type: Cat         |
| Name: Manny    |      | Name: Manny       |
| ...            |      | ...               |
+----------------+      +-------------------+
   ^         ^             ^
   |         |             |
someRef   otherRef      thirdRef
```

- `someRef` and `otherRef` both point to the same object. Same **identity**
- `thirdRef` points to a separate object with equal contents. **equality**

```
Cat* someRef, otherRef, thirdRef;
// ...

someRef == otherRef; // Shallow equal, same identity
someRef != thirdRef;

*someRef == *thirdRef; // Deep equal, invoke operator== on the Cat objects to check whethe
*someRef == *otherRef; // Obvious, they are the same objects
```

```
Cat someRef;
Cat otherRef;
Cat thirdRef;
// ...

someRef == otherRef; // Shallow equal; same identity
someRef != thirdRef;

someRef.equals(thirdRef); // Have to implement equals() to check for equality
```

22 Constants

- Implementation Principle: *Avoid magic numbers*
  - a magic number is a number literal or a string literal used directly in the code
  - No explanation why this number was chosen
  - No mechanism for changing the value
- Use a variable that has a descriptive name
- Better yet, use a *constant*

  - can not change
  - has a descriptive name
  - can be updated later (edit code)
  - can be replaced with a parameterised value (e.g. read a config file)

- Java `final`
  - Often combined with `static` to maintain a single object/variable across all object instances.
- C++ `const`
- C++ `#define` (discouraged but often seen, especially in old C code)

```
// #defines are replaced by the preprocessor
#define CURRENT_PLATFORM x86_64

// consts are more a promise by the user not to try
// to modify a variable. The compiler tries to warn you
// not to do anything stupid.
const int MAX_KITTENS=7;
const std::string DEFAULT_PROMPT="Press any key to continue...";
```

```
public class Rodent {
  private static final int MAXSIZE = 10; //cm
}
```

23 Summary

- Constructors to *set up objects* `X obj = new X()`
- Copy constructor to *clone an object* `X clone = original`
- Destructors to *clean up after objects* `delete x` and `delete [] x`
- Objects and references
- Objects and pointers
- Pass by Value vs Pass by Reference
- C++ Language extension: Operator Overloading
  - `Operator=()`
  - `Operator==()`
  - `Operator<<()`
- Implementation Principle: *Avoid magic numbers*
  - `const`
  - `static final`

24 Next Lecture: Exceptions and File IO

- Barnes & Kölling Chapter 9: Well Behaved Objects
- Barnes & Kölling Chapter 14: Handling Errors
- Barnes & Kölling Chapter 14.9: File-Based input/output
- Testing for Runtime Errors
- Runtime Errors vs Compile time Errors
- Input Sanitisation
- Error Reporting vs Error Handling
- Exception Handling
- File IO