Lecture 9
Constructor, Exceptions, Templates
TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
September 27th, 2023

IDA, Linköping University

9.1

Content

Contents

1 Constructor, copying and moving **1**
1.1 Why constructor? 1
1.2 Default constructor 2
1.3 Type conversion constructor 2
1.4 Copy constructor 3
1.5 Copy assignment operator 4
1.6 Move semantic 5

2 Exceptions **8**

3 Templates **10**

9.2

1 Constructor, copying and moving

1.1 Why constructor?

Initialization vs Assignment

- Initialisation is fundamental in C++ and different from assignment
  - **Initialisation** transforms an object initial garbage into valid data.
    - Defined in class with **constructor**
  - **Assignment** replace existing valid data with other valid data
    - Defined in class with **assignment operator**

```
// Initialisation: Default constructor
Widget x;
// Initialisation: copy constructor
Widget y(x);
// Initialisation: copy constructor (alternative form)
Widget z = x;
// Assignment: copy/assignment operator
z = x;
```

9.3

Note

- It is not always necessary to define all kinds of constructors and assignment operators. If you do not, the compiler will create a default version for you.

9.4

1.2 Default constructor

Default constructor

```
ArrayList() = default;
```

- A default constructor is a constructor that is called without arguments

```
ArrayList list;              // variable declaration without initialisation
```

- `= default` means that the constructor is generated by the compiler
  - members with a basic type are initialised with the value used in their declaration
  - members of class type are either initialised with arguments if specified in their declaration, or with their default constructor.
- Non-static members are initialised with their *NSDMI* ("non-static data member initialiser")

```
int m_size = 0;
int m_capacity = 10;
int* m_elements = new int[m_capacity];
```

9.5

Member initialisation list

A default constructor could be written as

```
ArrayList()
    : m_size{ 0 }, m_capacity { 10 }, m_elements { new int[m_capacity] }
{}
```

- If members are not in the list, they are initialised with the value specified in the definition or with their default value
- Member initialisation list lets us initialise (not assign) data members when we initialise our object

```
// Assignement
struct Widget {
        const int value;
        Widget();
};

Widget::Widget() {
       value = 42; //ERROR
}

// Initialisation
struct Widget {
    const int value;
    Widget();
};

Widget::Widget()
    : value{42}
{}
```

9.6

1.3 Type conversion constructor

Type conversion constructor

```
ArrayList::ArrayList(const vector<int>& v) {
    for (auto vi : v) {
        add(vi);
    }
}
```

- A constructor that can be called with the *argument of another type* is a type conversion constructor

```
ArrayList list(vector<int> {23, 24, 25, 26});
```

  - the syntax above is called direct initialisation
- The following syntax is called copy initialisation

```
ArrayList a1 = a2;  // same type - copy constructor does the initialisation
ArrayList a3 = vector<int>{1, 2, 3}            // implicit type conversion
ArrayList a4 = ArrayList(vector<int>{1, 2, 3}) // explicit type conversion
```

  - *move constructor* does the initialisation in the later two cases
  - optimization may occur. . .
- All explicit type conversion may occurs with this constructor, for example

```
auto a5 = static_cast<ArrayList>(vector<int> {23, 24, 25, 26});
```

9.7

1.4 Copy constructor

Copy constructor

In C++ a copy initialisation can happen in three cases:

1. A variable is created as a copy of an existing one

```
MyClass one;
MyClass two = one;
```

The previous code is equivalent to

```
MyClass one;
MyClass two(one);
```

2. Passing a variable as an argument to a function

```
void myFunction(MyClass arg) {
    ...
}
MyClass mc;
myFunction(mc);
```

3. An object is returned as the value of a function

```
MyClass myFunction() {
    MyClass mc
    return mc;
}
```

9.8

Copy constructor

Copy in C++ happens with the copy constructor:

- The syntax of a copy constructor is a constructor that takes a single parameter of the type using a const reference

```
class MyClass {
    public:
        MyClass();
        ~MyClass();
        MyClass(const MyClass& other); // Copy constructor
        /* ... */
};
```

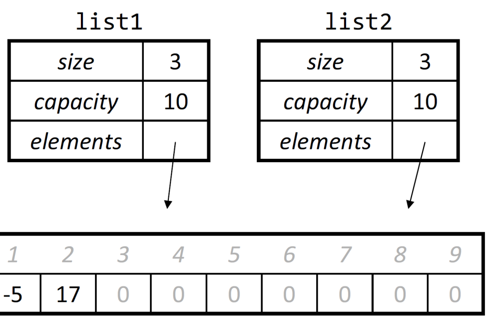
- The compiler can generate a default version of the copy constructor...

9.9

Common copy bug

- Copy initialisation of our ArrayList causes problem

```
ArrayList list1;
list1.add(42);
list1.add(-5);
list1.add(17);
ArrayList list2 = list1;
```



- A change in the list affects the other variable (bad!)

```
list2.add(88);
list1.remove(0);
```

  – When the objects are destroyed, the memory is deleted twice (bad!)

9.10

Deep copy

- To fix the copy bug, we need to write a constructor that makes a *deep copy* of ArrayList

- **Rule of Three**: If a class has one of the three following member function:
  – Destructor
  – Copy constructor
  – Copy assignment operator

  it should probably have all three of them.

9.11

Prevent copy

- A simple solution is to disable the copy constructor:

```
// ArrayList.h
ArrayList(const ArrayList& list) = delete;
```

  - Now attempts to do a copy will lead to an error
  - It solves the problem but it is too restricted

9.12

Code for copy constructor

```
// ArrayList.cpp
ArrayList::ArrayList(const ArrayList& other) {
    m_capacity = other.m_capacity;
    m_size = other.m_size;
    m_elements = new int[m_capacity]; // deep copy
    std::copy(other.m_elements, other.m_elements + m_size, m_elements);
}
```

9.13

1.5 Copy assignment operator

Copy assignment operator

Assignment in C++ is different from the initialisation and only takes place if an existing object is explicitly assigned a new value:

```
MyClass one, two;
two = one;
```

- Compare to the following where `two` is initialised as a copy of `one`

```
MyClass one;
MyClass two = one;
```

9.14

Copy assignment operator

Copy assignment in C++ is done by the copy operator:

- Syntactically, the copy assignment operator is more complex than the copy constructor:

```
class MyClass {
    public:
        MyClass();
        ~MyClass();
        MyClass(const MyClass& other); // Copy constructor
        MyClass& operator= (const MyClass& other); // Assignment operator
        /* ... */
};
```

- The compiler-generated copy assignment operator works only for simple cases. . .

9.15

Code for deep copy

The code for a correct copy assignment operator is more involved than the copy constructor.

- To some extend C++ allows for maximal flexibility. For example, the following is a valid implementation:

```
void MyClass::operator= (const MyClass& other) {
    cout << "I’m␣sorry,␣Dave.␣I’m␣afraid␣I␣can’t␣copy␣that␣object." << endl;
}
```

9.16

Code for deep copy assignment: version 1

```
/* Many common errors. Do not use as reference! */
void ArrayList::operator= (const ArrayList& other) {
    m_capacity = other.m_capacity;
    m_size = other.m_size;
    m_elements = new int[m_capacity]; // deep copy
    std::copy(other.m_elements, other.m_elements + m_size, m_elements);
}
```

- Code is based on the copy constructor
  - However, when the copy operator is called ArrayList already has an allocated array of elements, which lead to memory leak. . .

9.17

Code for deep copy assignment: version 2

```
/* Many common errors. Do not use as reference! */
void ArrayList::operator= (const ArrayList& other) {
    delete[] m_elements;

    m_capacity = other.m_capacity;
    m_size = other.m_size;
    m_elements = new int[m_capacity]; // deep copy
    std::copy(other.m_elements, other.m_elements + m_size, m_elements);
}
```

- All code after the **delete**[] is the same has with copy constructor
  - No coincidence— in most cases, there is a large overlap between the two operation
  - Since we cannot call our own copy constructor directly (or any other constructor), we avoid the code duplication using a helper function

9.18

Code for deep copy assignment: version 3

```
void ArrayList::copyOther(const ArrayList& other) {
    m_capacity = other.m_capacity;
    m_size = other.m_size;
    m_elements = new int[m_capacity]; // deep copy
    std::copy(other.m_elements, other.m_elements + m_size, m_elements);
}

ArrayList::ArrayList(const ArrayList& other) {
    copyOther(other);
}

/* Not completely perfect yet. Do not use as reference! */
void ArrayList::operator= (const ArrayList& other) {
    delete[] m_elements;
    copyOther(other);
}
```

- We have a few things left to consider
  - Consider the following:

```
ArrayList one;
one = one;
```

9.19

Code for deep copy assignment: version 4

```
/* Not completely perfect yet. Do not use as reference! */
void ArrayList::operator= (const ArrayList& other) {
     if (this != &other) {
     delete[] m_elements;
     copyOther(other);
     }
}
```

- A last bug to take care of
- Consider the following:

```
ArrayList one, two, three;
three = two = one;
```

9.20

Code for deep copy assignment: final version

```
ArrayList& ArrayList::operator= (const ArrayList& other) {
    if (this != &other) {
        delete[] m_elements;
        copyOther(other);
    }
    return *this;
}
```

9.21

1.6 Move semantic

Before C++11

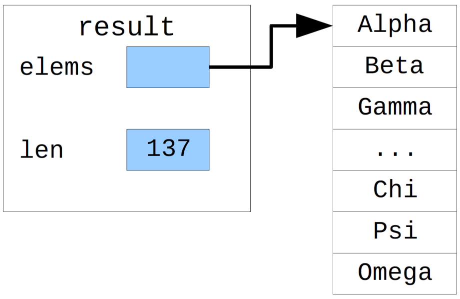
```
vector<string> ReadAllWords(const string& filename) {
    ifstream input(filename.c_str());

    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

- How effective is that code?

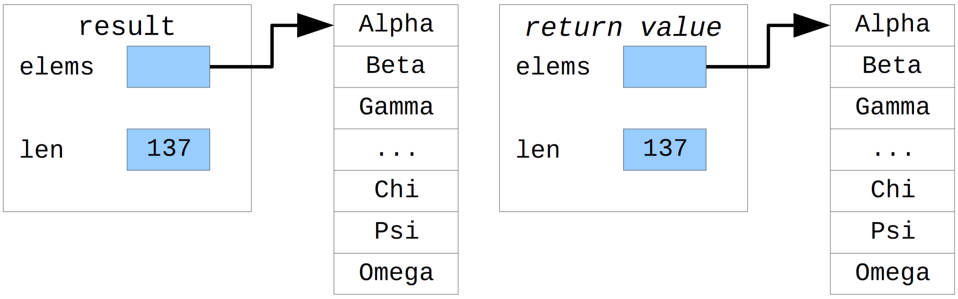
9.22

Before C++11



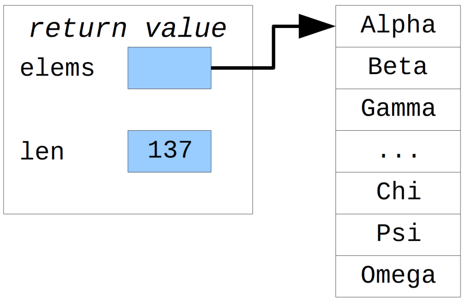
9.23

Before C++11



9.24

Before C++11



9.25

After C++11

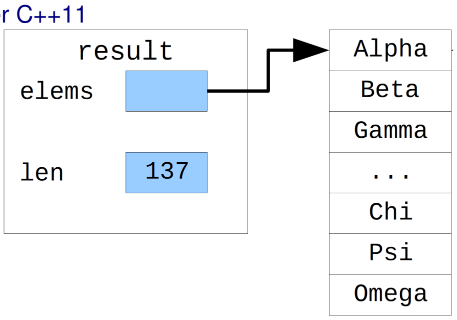
```
vector<string> ReadAllWords(const string& filename) {
    ifstream input(filename.c_str());

    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

- No change in code. . .

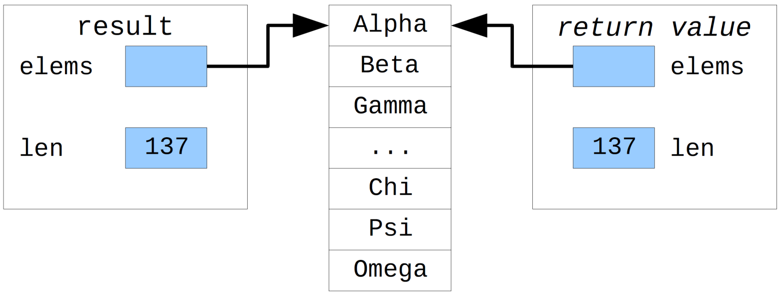
9.26

After C++11



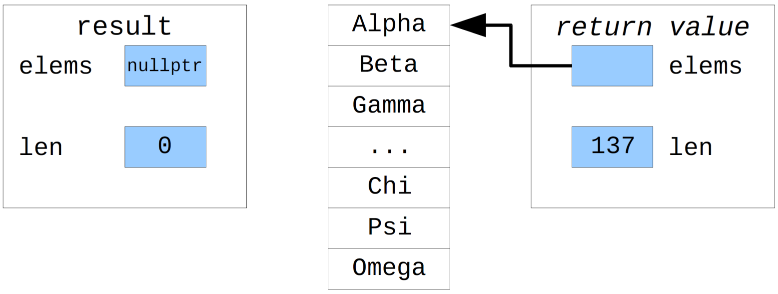
9.27

After C++11



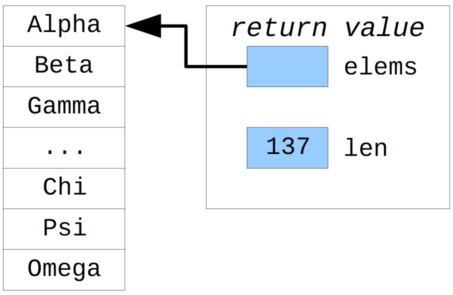
9.28

After C++11



9.29

After C++11



9.30

Move semantic

- Copy semantic (C++03): Can duplicate an object
  - Copy constructor and copy assignment operator
- Move semantic (C++11): Can move an object to an other one
  - Move constructor and move assignment operator
- Move semantic gives better performance in most cases
- Copy and move constructors are sometimes avoided completely with copy-elision[1], e.g. `T x = T(T(T()));`

9.31

Rvalue-reference

- Syntax `Type &&`
- Reference to a temporary expression
- Represents an expression that can be moved

9.32

[1] https://en.cppreference.com/w/cpp/language/copy_elision

With C++11

```
/* Move constructor */
ArrayList::ArrayList(ArrayList&& other) {
    m_elements = other.m_elements;
    m_size = other.m_size;
    m_capacity = other.m_capacity;
    other.m_size = 0;
    other.m_capacity = 10;
    other.m_elements = new int[other.m_capacity];
}

/* Move operator */
ArrayList& ArrayList::operator= (ArrayList&& other) {
    if (this != &other) {
        delete[] m_elements;
        m_elements = other.m_elements;
        m_size = other.m_size;
        m_capacity = other.m_capacity;
        other.m_size = 0;
        other.m_capacity = 10;
        other.m_elements = new int[other.m_capacity];
    }
    return *this;
}
```

9.33

Move semantic $\neq$ copy semantic

- Returns the object in the same way
- C++11 tries to move first otherwise it fallbacks to copy
- Objects can be moveable even if not copyable

9.34

Rule of five

- Implicit definition of a move constructor or a move assignment operator is prevented by the presence of user defined
  - destructor, or
  - copy constructor or,
  - copy assignment operator.
- **Rule of Five**: If a class has one of the five following member functions:
  - Destructor
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - move assignment operator

  it should probably have all five of them.

9.35

2 Exceptions

Exceptions

Problem: size vs capacity

- What happens if the client access an element at a position after the size? `list.get(7)`

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	9	7	5	0	0	0	0	0
size	5	*capacity*	10							

  - Without a check this is allowed and returns a 0
    - Is this good or bad? What (if possible) could we do about it?

9.36

Error messages and return values

- Error printout
  - Print an error message and exit the program
    - A bit drastic, the program get no chance to recover
- Return values
  - Return a special value indicating that something went wrong, t.ex. -1
    - The problem is that all integers can exist in the list! How to distinguish normal value from error value?

9.37

Preconditions

- **preconditions**: make assumption in your code that a condition is *true*.
  - Often documented as a comment:

```
/*
 * Returns the element at the given index.
 * Precondition: 0 <= index < size
 */
int ArrayList::get(int index) {
    return m_elements[index];
}
```

  - Having a documented precondition does not "solve" the problem, but it warns the user.
  - But what if the user does not read the documentation (or ignore it) and access a value at a bad index?
  - Can we ensure that the user *must* follow the precondition?

9.38

Throw Exceptions

- `throw expression;`
  - Generates an exception that aborts the program if there is no handling of the exception (`catch`)
  - In Java, only objects inheriting `Exceptions` can be thrown; in C++ all types can be thrown (int, string, etc.)

```
throw 0;                    // Throw an int
throw new vector<double>;   // Throw a vector<double> *
throw 3.14159;              // Throw a double
```

  - There is a class `std::exception` that can be used

```
try {
    // Do something
}
catch(int myInt) {
    // If the code throw an int the execution continues here
}
catch(const exception& e) {
    // If the code throw a std::exception the execution continues here
    cout << "exception:␣" << e.what() << '\n';
}
catch(...) { // Special syntax to catch all remaining exceptions
    cout << "An␣unexpected␣error␣has␣ocurred!\n";
}
```

9.39

Standard exceptions

				kastas t.ex. av:
`exception`				
	`logic_error`			
		`domain_error`		*otillåtna funktionsvärden*
		`invalid_argument`		`bitset`-*konstruktor*
		`length_error`		*objektlängd överskrids*
		`out_of_range`		`at()`
		`furure_error`		*funktioner i trådbiblioteket*
	`runtime_error`			
		`range_error`		*vissa beräkningar*
		`overflow_error`		`bitset::to_long()`
		`underflow_error`		*vissa beräkningar*
		`system_error`		*systemrelaterade fel*
			`ios_base::failure`	`ios_base::clear()`
	`bad_typeid`			**`typeid`**
	`bad_cast`			**`dynamic_cast`**
	`bad_weak_pointer`			`shared_ptr`-*konstruktorer*
	`bad_exception`			*exception specification*
	`bad_function_call`			`function::`**`operator`**`()`
	`bad_alloc`			**`new`**
		`bad_array_new_length`		**`new`**`[]`

9.40

Private helper function

```
// In ArrayList.h
private:
    void checkIndex(int i, int min, int max) const;

// In ArrayList.cpp
void ArrayList::checkIndex(int i, int min, int max) const {
    if (i < min || i > max) {
        throw std::out_of_range("Index␣" + std::to_string(i)
                              + "␣out␣of␣range;␣(must␣be␣between␣"
                              + std::to_string(min) + "␣and␣"
                              + std::to_string(max) + ")");
    }
}
```

9.41

Exceptions and dynamic memory

```
void f(){
  if(std::rand() > RAND_MAX / 2) throw std::random_exception();
}


try{
  int* a = new int[10];
  ArrayList b {10, 2, 4, 5};
  f();
  delete[] a;
} catch(const std::random_exception& re){
}
```

Without proper care, exceptions can lead to memory leaks!

9.42

3 Templates

What is a template?

- A *template function* is a model to generate function
- It is equivalent of letting the compiler generate each function automatically for each type
- Instantiation of a template happens when a given template function is called for a specific type

9.43

What is a template?

- To declare a *template function*, just add the following line in front of a function definition:

```
template <typename T>
```

- `T` is a template parameter which will be replaced by a specific type when you use the template function. The function cannot be used with a type called `T`.

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

9.44

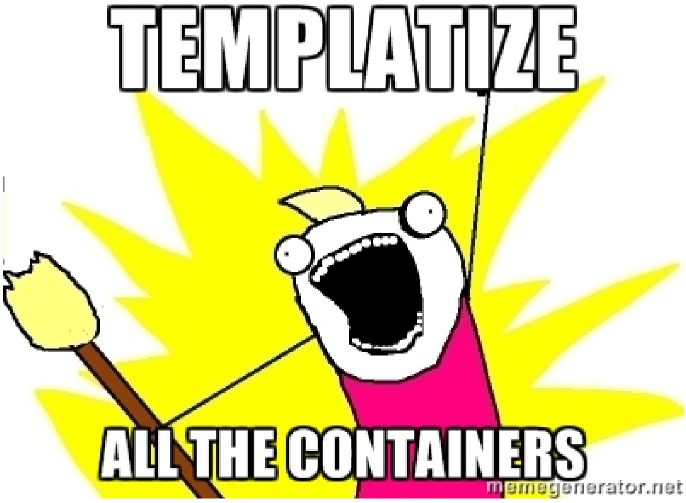
Validation of templates

- Compiler “verifies” that template can be instantiated.
- Template functions can be instantiated only if all the operations on the variables used in the function are valid.

9.45

Usage of templates

- In this lecture, we only use templates to avoid rewriting functions many times.
- Templates are much more powerful and useful (and complicated).



9.46

Class template

- Mark each class/function as template in .h- and .cpp-files
- Replace the previous type (t.ex. **int**) with T in code

```
// ClassName.h
template<typename T>
class ClassName {
    ...
};

// ClassName.cpp
template<typename T>
type ClassName::name(parameters) {
    ...
}
```

9.47

.h and .cpp for template class

- In C++ template system, as soon as the compilers sees templates' being used with a given type, it needs to see the definition (and not only the declaration).
  - Either write all code in .h-file,
  - or include .cpp-file at the end of .h-file.

```
// ClassName.h
#ifndef _classname_h
#define _classname_h

template<typename T>
class ClassName {
    ...
};

#include "ClassName.cpp"
#endif   // _classname_h
```

9.48