# Introduction to the C++ Programming Language

## Day 3

**Jonas Rylund Glesaaen**
jonas@glesaaen.com

**September 30th 2015**

# What will we learn?

- ~~Basic C++ syntax~~
- ~~Control structures~~
- ~~Functions~~
- Structs and classes (today and Thursday)
- Templates and STL (Thursday and Friday)
- Exceptions (Friday)

# Today's topics

1. Dynamic Memory Management
2. Object Oriented Programming
3. Programming Practices
4. Recap

# Dynamic Memory Management

# Disclaimer

# Compile time vs runtime

**Compile time** :

Things known/decided when the program is compiled, before it is ever run, also known at runtime

**Runtime** :

Things known/decided as the program is run, not known at compile time

# Two types of memory management

So far we have programmed using automatic memory management:

- The memory need of every individual object is known at compile time

```
double array[100];
```

Known by the compiler, is a constant expression

- Cannot change the size after initialisation

# Two types of memory management

Managing memory at runtime is called dynamic memory management

## Advantage:

- More flexible than automatic memory management

## Disadvantage:

- Compiler cannot optimise your code as well

# Constructing memory at runtime

Use **new** expressions to create memory

`new` type { constructor arguments };

The expression returns a pointer to the memory location that was created

Memory allocated this way is not limited by its scope
{but the associated pointer is}

# Constructing memory at runtime

C++11 Style
Constructor Braces

Use new expressions to create memory

`new` type { constructor arguments } ;

The expression returns a pointer to the memory location that was created

Memory allocated this way is not limited by its scope
{but the associated pointer is}

# Constructing memory at runtime

```
int * iptr;
iptr = new int {5};

auto dptr = new double {5.1};
*dptr = 2.16;
```

# Constructing memory at runtime

Might seem a bit pointless at the moment but it will be more important as we later discuss polymorphism and data ownership

It also adds flexibility to our programming

# No more free cleanup

Dynamic memory isn't limited by scope

So how is it cleaned up when it is no longer needed?

# No more free cleanup

Dynamic memory isn't limited by scope

So how is it cleaned up when it is no longer needed?

We have to do it ( ;_; )

# No more free cleanup

## A fun little program

```
int main()
{
  while (true) {
    new int {0};
  }
}
```

# No more free cleanup

Clean up memory with delete expressions

```
delete pointer to memory ;
```

new and delete must always come in pairs, otherwise you have memory leaks

# No more free cleanup

```cpp
int main()
{
  auto dynamic_memory = new int {4};

  // Carry out the program

  delete dynamic_memory;
}
```

# Creating arrays

## More immediate value with arrays

```
auto array = new int [10];
```

Does not need to be a
compile time constant

# Creating arrays

## More immediate value with arrays

```
auto array = new int [10];

// ...

delete [] array;
```

# Creating arrays

```cpp
int main()
{
  unsigned size {1};
  std::cout << "Array size: ";
  std::cin >> size;

  auto array = new int [size];

  //...
  delete [] array;
}
```

# Multi dimensional arrays

Can also do dynamic multi dimensional arrays

Multi dimensional arrays are just arrays of arrays

Type of 2D **int** array: `int**`

Type of 4D **float** array: `float****`

# Multi dimensional arrays

```
float** createArray(unsigned size_x, unsigned size_y)
{
  auto array = new float* [size_x];

  for (auto i = 0; i < size_x; ++i) {
    array[i] = new float [size_y];

    for (auto j = 0; j < size_y; ++j) {
      array[i][j] = 0.;
    }
  }

  return array;
}
```

# Multi dimensional arrays

```
void deleteArray(float ** array, unsigned size_x)
{
  for (auto i = 0; i < size_x; ++i) {
    delete [] array[i];
  }

  delete [] array;
}
```

# Welcome to Memory Leak City

```
population += you;
```

If you believe that you are able to perfectly manage your own memory you are **wrong** {I've tried as well}

There are so many things that can go wrong

We will look at some options in the exercises and on the final day

# Debugging memory leaks

There are many tools available for debugging memory issues, but one of the best known ones is valgrind

Valgrind checks for things such as

- Reading out of bounds
- Using undefined values
- Double freeing of memory
- Memory leaks

# Object Oriented Programming

# What exactly is OO?

Object oriented programming is a programming paradigm where one focuses on objects rather than methods.

One organises the code into objects and interfaces, defining how they interact with each other and how they can be manipulated.

# What exactly is OO? - example

An Address Book is

- made up of addresses

And one can manipulate it in multiple ways

- Add addresses
- Remove addresses
- Search for addresses
- Copy your friends' books

# What exactly is OO? - example

```
class AddressBook
{
public:
  void addAddress(Address new_address);
  void removeAddress(int address_id);
  Address search(std::string search_string) const;
  void copyAddresses(const AddressBook & other_book);

private:
  std::list<Address> addresses;
};
```

# Aggregate containers

Can use the struct keyword to declare a new type that is a combination of other types

```
struct Address
{
  std::string name;
  std::string street_name;
  unsigned street_number;
  unsigned zip_code;
};
```

This is a great organisational tool to help express yourself when writing code

# Aggregate containers

Can use the struct keyword to declare a new type that is a combination of other types

```
struct Address
{
  std::string name;
  std::string street_name;
  unsigned street_number;
  unsigned zip_code;
};
```

Don't forget the semicolon

This is a great organisational tool to help express yourself when writing code

# Aggregate containers

```
struct Coordinate
{
  double x, y, z;
};

int main()
{
  Coordinate edge;
  edge.x = 4.5;
  edge.y = 0.0;
  edge.z = 9.1;

  //...
}
```

Type declaration

# Aggregate containers

```
struct Coordinate
{
  double x, y, z;
};

int main()
{
  Coordinate edge {4.5, 0.0, 9.1};

  //...
}
```

Construction with initialiser list
Order as in type declaration

# Class members - variables

The variables in type declarations are called member variables

When accessed they can be used as any other variable of the same type

```
int main()
{
  Coordinate endp {9.4, 8.2, -3.4};
  auto dist = norm(endp.x, endp.y, endp.z);
}
```

# Class members - variables

The variables in type declarations are called member variables

When accessed they can be used as any other variable of the same type

```
int main()
{
  Coordinate endp {9.4, 8.2, -3.4};
  auto dist = norm(endp.x, endp.y, endp.z);
}
```

Could overload `norm` to accept a `Coordinate` object

# Class members - functions

**The Real™ Object Oriented Programming starts here**

# Class members - functions

## Classes can also have functions as members

```
class Coordinate
{
public:
  double x, y, z;

  double norm() const
  {
    return std::sqrt(x*x + y*y + z*z);
  }
};
```

## The member variables are "in scope" of the member functions

# Class members - functions

## The function is bound to the object calling it

```
int main()
{
Coordinate edge {4, 2, 1};
Coordinate point {-1, 6, 2};

double length = edge.norm(); ←——— 4.58...
double distance = point.norm(); ←——— 6.40...
}
```

# Declaration and definition

## Special syntax for defining member functions

```
class Coordinate
{
public:
  double x, y, z;

  double norm() const
  {
    return std::sqrt(x*x + y*y + z*z);
  }
};
```

Automatically inline function

# Declaration and definition

## Special syntax for defining member functions

```
class Coordinate
{
public:
  double x, y, z;

  double norm() const;
};

double Coordinate::norm() const
{
  return std::sqrt(x*x + y*y + z*z);
}
```

# Encapsulation

Encapsulation is the concept of separating the outward functionality of a class from the inner workings of it

# Encapsulation - example

```
class Coordinate
{
private:
  //...

public:
  void setCartesian(double x, double y);
  void setPolar(double r, double phi);
  double norm() const;
};
```

It is not important for someone using the class whether the coordinate is stored in the polar or Cartesian coordinate system

# Encapsulation - example

Implementation #1

```
class Coordinate
{
private:
  double radius, angle;

public:
  void setCartesian(double x, double y)
  {
    radius = std::sqrt(x*x + y*y);
    angle = std::atan(y/x);
  }

  void setPolar(double r, double phi)
  {
    radius = r;
    angle = phi;
  }

  double norm() const
  {
    return radius;
  }
};
```

Implementation #2

```
class Coordinate
{
private:
  double x_comp, y_comp;

public:
  void setCartesian(double x, double y)
  {
    x_comp = x;
    y_comp = y;
  }

  void setPolar(double r, double phi)
  {
    x_comp = r * std::cos(phi);
    y_comp = r * std::sin(phi);
  }

  double norm() const
  {
    return std::sqrt(
      x_comp*x_comp + y_comp*y_comp);
  }
};
```

# Encapsulation - example

For someone using the Coordinate class
these two implementations are identical

Thus we can switch between the two without
worrying that the rest of our program will change

Encapsulation adds flexibility

# Access levels

Class access levels facilitate encapsulation in C++

There are 3 access levels in C++

- **public**
  Accessible by everyone
- **private**
  Only accessible by other members and friends
- **protected**
  Accessible by children classes

# Access levels

## Cannot access private members from the outside

```
class Coordinate
{
private:
  double x, y;
};

int main()
{
  Coordinate c;
  c.x = 5;
}
```

This will not compile

# class vs struct

The only difference between class and struct in C++

- Classes are private by default
- Structs are public by default

```
struct Container
{
  double x;
};
```

x is public

# class vs struct

The only difference between class and struct in C++

Classes are private by default

Structs are public by default

```
class Container
{
  double x;
};
```

x is private

# class vs struct

For readability one makes the distinction anyway

**struct**
congregate data structure

**class**
encapsulated type with an interface

# Friendship

## friends of classes can access their private members

```
class Coordinate
{
friend double norm(const Coordinate &);

private:
  double x, y;
};

double norm(const Coordinate & c)
{
  return std::sqrt(c.x*c.x + c.y*c.y);
}
```

# Constructor

**The constructor is the function that is called when the object is initialised**

**There are 3 types of default constructors**

- **Default constructor**
  Calls the default constructor on all members
- **Copy constructor**
  Copies all nonstatic members
- **Move constructor** {C++11}

# Constructor

```
class Coordinate
{
private:
  double x, y;
};

int main()
{
  Coordinate c1;               // Default constructor

  Coordinate c2 {c1};          // Copy constructor

  Coordinate c3 = c2;          // Copy constructor

  auto c4 = Coordinate {};     // Default constructor
}
```

# Constructor

## We can change their behaviour

```
class Coordinate
{
public:
  Coordinate()
  {
    x = 0.0;
    y = 0.0;
  }

private:
  double x, y;
};
```

Should always initialise
built in types with a default value

# Constructor

## We can change their behaviour

```
class Coordinate
{
public:
  Coordinate()
    : x {0.0},
      y {0.0}
  {}

private:
  double x, y;
};
```

# Constructor

## ...or we can declare new constructors

```
class Coordinate
{
public:
  Coordinate(double x0, double y0)
    : x {x0},
      y {y0}
  {}

private:
  double x, y;
};
```

# Constructor

## ...or we can declare new constructors

```
class Coordinate
{
public:
  Coordinate(double x0, double y0)
    : x {x0},
      y {y0}
  {}

private:
  double x, y;
};
```

Note: if you declare your own constructor, the default constructor will not be automatically generated any more

# Constructor

```
int main()
{
  Coordinate c1 {5.2, 9.1};

  Coordinate c2;
}
```

Calls our new constructor

Error: no such constructor

# Constructor - default

## but we can reinstate the default constructors

```
class Coordinate
{
public:
  Coordinate(double x0, double y0)
    : x {x0},
      y {y0}
  {}

  Coordinate() = default;
  Coordinate(const Coordinate &) = default;

private:
  double x, y;
};
```



{C++11}

# Constructor - delete

## and we can delete them if we don’t want them

```
class Coordinate
{
public:
  Coordinate(double x0, double y0)
    : x {x0},
      y {y0}
  {}

  Coordinate(const Coordinate &) = delete;
  Coordinate(Coordinate &&) = delete;

private:
  double x, y;
};
```



{C++11}

# Implicit conversions

## A constructor taking only one argument can be used by the compiler for conversions

```
class SomeClass
{
public:
  SomeClass(double);
};

void someFunction(SomeClass);

int main()
{
  someFunction(2.45);
}
```

# Implicit conversions

## These can be disabled by the explicit keyword

```
class SomeClass
{
public:
  explicit SomeClass(double);
};
```

# Implicit conversions

**Pitfall:**
**Constructors with default arguments can also be used**

```
class SomeClass
{
public:
  explicit SomeClass(double, double = 2.4);
};
```

# lvalues and rvalues

$\boxed{\text{lvalue}} = \boxed{\text{rvalue}}$

# lvalues and rvalues

## lvalue

An lvalue is an object that persists after a single expression, can be at the left hand side of an assignment operator

## rvalue

An rvalue is a temporary object that do not persist after the expression, can only be at the right hand side of an assignment operator

# lvalue and rvalue references

**Normal references are lvalue references**

`type &`

**rvalue references are a way to signal that we don't intend to use the object after that point**

`type &&`

**Convert lvalue reference to rvalue reference with the** `std::move` **function in** `<utility>`



{C++11}

# Move constructors

```
class MemoryManager
{
public:
  MemoryManager(const MemoryManager & copy)
    : d_ptr {new double {*(copy.d_ptr)}} {}

  MemoryManager(MemoryManager && move)
    : d_ptr {move.d_ptr}
  {
    move.d_ptr = nullptr;
  }

private:
  double * d_ptr;
};
```



{C++11}

# Destructor

The destructor is the function that is called when the object goes out of scope

It will always automatically call the destructor of all the class' members, but you can add additional functionality

# Destructor - example

```
class MemoryManager
{
public:
  MemoryManager() = default;
  MemoryManager(const MemoryManager &);

  ~MemoryManager()
  {
    delete d_ptr;
  }

private:
  double * d_ptr;
};
```

# Operator overloading

It is also possible to define how your class behaves together with all the operators of C++

What should `Coordinate+Coordinate` do?

What should `Coordinate*Coordinate` do?

What about `++Coordinate`?

# Operator overloading - arithmetic

```
class Coordinate
{
public:
  Coordinate operator+(const Coordinate & rhs) const
  {
    auto result = *this;
    result.x += rhs.x;
    result.y += rhs.y;
    result.z += rhs.z;

    return result;
  }
};
```

For member function declarations the calling object is always on the left hand side of the operator for two variable operators

# Operator overloading - arithmetic

```
class Coordinate
{
public:
  Coordinate operator+(const Coordinate & rhs) const
  {
    auto result = *this;
    result.x += rhs.x;
    result.y += rhs.y;
    result.z += rhs.z;

    return result;
  }
};
```

**this is a pointer to the object calling the function**

For member function declarations the calling object is always on the left hand side of the operator for two variable operators

# Operator overloading - arithmetic

```
class Coordinate
{
friend Coordinate operator+(
  const Coordinate &, const Coordinate &);
};

Coordinate operator+(
  const Coordinate &lhs, const Coordinate &rhs)
{
  auto result = lsh;
  result.x += rhs.x;
  result.y += rhs.y;
  result.z += rhs.z;

  return result;
}
```

# Order matters

## Just as in mathematics, argument order matters

```
class Coordinate
{
public:
  Coordinate operator*(double);
};

int main()
{
  Coordinate distance {4.5, 9.0};

  auto twice = distance * 2;      ← OK
  auto thrice = 3 * distance;     ← Error: not defined
}
```

# Operator overloading - assignment

## The assignment operator is also automatically generated by the compiler if not explicitly declared

```
class Coordinate
{
public:
  Coordinate& operator=(const Coordinate &);
  Coordinate& operator=(Coordinate &&);
};
```

## Should return a reference to this

# Operator overloading - stream

Change printing behaviour by overloading the bitshift operator for stream objects

# Operator overloading - stream

```
class Coordinate
{
friend std::ostream& operator<<(
  std::ostream&, const Coordinate &);
};

std::ostream& operator<<(
  std::ostream & out, const Coordinate & c)
{
  out << "{" << c.x << ","
    << c.y << "," << c.z << "}";

  return out;
}
```

# Pre- and post increment

## Pre increment `++object`

Returns the value the object has after it has been incremented

## Post increment `object++`

Returns the value the object had before it was incremented

# Operator overloading - increment

```
class Counter
{
public:
  Counter& operator++()
  {
    ++count;
    return *this;
  }

  Counter operator++(int)
  {
    auto before = *this;
    ++count;
    return before;
  }

private:
  unsigned count;
};
```

Pre increment

# Operator overloading - increment

```
class Counter
{
public:
  Counter& operator++()
  {
    ++count;
    return *this;
  }

  Counter operator++(int)
  {
    auto before = *this;
    ++count;
    return before;
  }

private:
  unsigned count;
};
```

Post increment

# And many more...

All the other operators can be overloaded as well

- Reference & and dereference *
- Arithmetic assignment `+=  -=  *=  /=`
- Call operator ()
- Element operator []
- Cast operator

except for member access .

# const-ness

## Member functions that leave the object unchanged should be marked const

```
class Coordinate
{
public:
  double norm() const;
};
```

## Constant instances of a class can only call methods that are marked const

# static methods in classes

Just as with functions, a static member of a class transcends the individual class instances

Static methods can only use static variables and call other static methods

Access static methods as if it was in a namespace

```
SomeClass::staticMethod();
```

# static methods in classes

```
class InstanceCounter
{
public:
  InstanceCounter() { ++count; }
  ~InstanceCounter() { --count; }

  static unsigned getCount()
  {
    return count;
  }

private:
  static unsigned count;
};
```

# static methods in classes

```
unsigned InstanceCounter::count = 0;

int main()
{
  InstanceCounter i1;
  InstanceCounter::getCount();
  {
    InstanceCounter i1, i2, i3, i4;
    InstanceCounter::getCount();
  }
  InstanceCounter::getCount();
}
```

# static methods in classes

```
unsigned InstanceCounter::count = 0;

int main()
{
  InstanceCounter i1;
  InstanceCounter::getCount();          1
  {
    InstanceCounter i1, i2, i3, i4;
    InstanceCounter::getCount();        5
  }
  InstanceCounter::getCount();          1
}
```

# The rule of ~~three~~ five

It is generally a good idea to explicitly define

- Destructor
- Copy constructor
- Move constructor {C++11}
- Copy assignment operator
- Move assignment operator {C++11}

# Programming Practices

# Good Programming Practices

- Be wary of memory leaks
- Continue to use `const` consistently
- Assign values to built in types at construction
- Respect encapsulation
- Follow the rule of five

# Recap

# Recap Day 3

- Use new and delete to manage memory dynamically
- Classes can have member variables and member functions
- Member functions are bound to the class instance
- Classes have three levels of access levels

# Recap Day 3

- Constructors manipulate how new objects are created
- Destructors define what happens when they go out of scope
- Operators can be overloaded to define how they work with your classes