TDDD38/726G82: Adv. Programming in C++

Language constructs and rules II

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 Compound Types
2 Operator Overloading
3 Value categories

Compound Types

`struct`

`struct` is inherited from C, but very common in C++ as well

```
struct Vector
{
  int x { 0 };
  int y { 0 };
};
```

Compound Types

`struct`

As we will see later on there are some difference between C and C++, but for now, they are the same thing

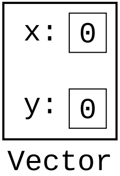
```
struct Vector
{
  int x { 0 };
  int y { 0 };
};
```

Compound Types

struct

A struct bundles variables together into one variable, usually called an *object*

```
struct Vector
{
  int x { 0 };
  int y { 0 };
};
```

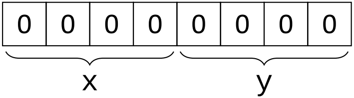


Compound Types

`struct`

> In memory this is represented by putting all *data members* in sequence, so the declared order of members determines their stored order

```
struct Vector
{
  int x { 0 };
  int y { 0 };
};
```



Compound Types

Classes and structs are the same thing!

```
struct Vector_Struct
{

  int x;
  int y;
};
```

```
class Vector_Class
{

  int x:
  int y;
};
```

What is the difference?

Compound Types

Classes and structs are the same thing!

```
struct Vector_Struct
{
public:
  int x;
  int y;
};
```

```
class Vector_Class
{
private:
  int x:
  int y;
};
```

Compound Types

`struct` vs. `class`

- There are exactly two functional differences between `struct` and `class`
- In `struct` every member is `public` by default
- While in `class` all members are `private` by default
- The second difference is similiar but related to inheritance (we'll talk about it later)
- Besides this they are *functionally* the same thing

Compound Types

Mental Model

- Both structs and classes are *compound* types, meaning they are constructed by storing multiple objects/variables
- These objects are called *data members* (sometimes called fields or instance variables)
- We think of data members as separate variables stored *inside* the class
- This is mainly how the compiler sees it as well
- Once our code has compiled, objects will just be a sequence of variables (specifically the data members)
- The data members will be stored in the same order as they are declared (this is *always* true: the compiler is not allowed to change the order)

Compound Types

Padding & Alignment

- All data types have a property called *alignment*
- A types alignment specifies an integer which each object's address must be *evenly divisible* by
- **Example:** It is common that `int` has alignment 4 which means each `int` must be located at an address which is a multiple of 4.

Compound Types

Padding & Alignment

- Alignment is important in order to efficiently utilize the architecture of the CPU (and memory units)
- Most modern CPUs have *aligned access* which means the hardware is designed to efficiently read values of certain sizes at certain *alignments*

Compound Types

Padding & Alignment

- class types consists of several data members (each with their own alignment)
- To make sure that the memory representation of objects is as efficient as possible the compiler has to make sure that the data member with the *largest* alignment will be properly aligned in all situations
- Because of this the class type will always have the same alignment as the data member with the largest alignment
- This can however lead to some wasted space (called *padding*)

Compound Types

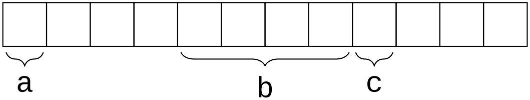
Padding & Alignment

```
struct X
{
  char a;
  int  b;
  char c;
};
```

Compound Types

Padding & Alignment

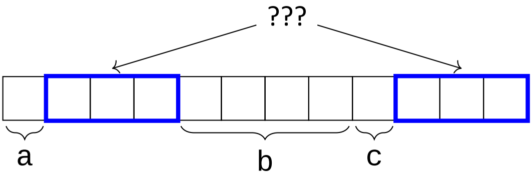
```
struct X
{
  char a;
  int  b;
  char c;
};
```



Compound Types

Padding & Alignment

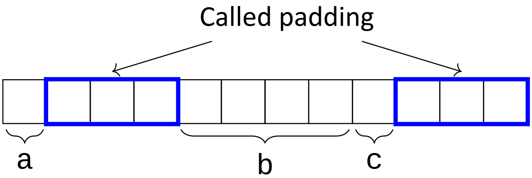
```
struct X
{
  char a;
  int  b;
  char c;
};
```



Compound Types

Padding & Alignment

```
struct X
{
  char a;
  int  b;
  char c;
};
```



Compound Types

Padding & Alignment

- In the previous (and next) example we assume that `char` has alignment 1 (meaning it can be stored on *any* address) while `int` has alignment 4 (meaning it must be stored on an address which is a multiple of 4)
- So X has alignment 4 (the largest alignment of all data members)
- The compiler **must** store all data members in their declared order
- Because of this, the compiler is forced to have 4 bytes *before* the `int`
- But we only really *need* 1 byte, so the compiler inserts 3 unused bytes

Compound Types

Padding & Alignment

- After the `int` we store another `char` meaning we have add one more byte
- This puts the total size of X at 9
- But what happens if we need to store objects of type X in an array?
- Then the objects must be placed at addresses which are multiples of 4 (since the alignment of X is 4)
- But this can never happen if the size is not evenly divisible by 4
- So the compiler extends the size to `12` (it adds 3 more unused bytes at the end)

Compound Types

Padding & Alignment

- All of these unused bytes are called *padding* and can be inserted by the compiler *before* any data member, as well as at the *end* of a struct/class
- However, we can control the padding *somewhat* by thinking about the order we store our data members in (see next example)
- A general rule of thumb is to sort your data members based on *size*
- The best method is to sort your data members in *descending* order (meaning you put the largest types first)

Compound Types

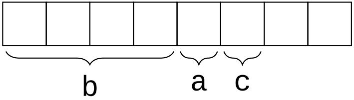
Padding & Alignment

```
struct X
{
  int  b;
  char a;
  char c;
};
```

Compound Types

Padding & Alignment

```
struct X
{
  int  b;
  char a;
  char c;
};
```

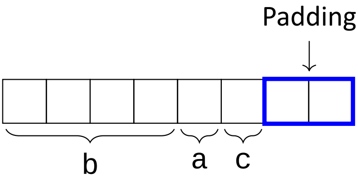


LINKÖPING UNIVERSITY

Compound Types

Padding & Alignment

```
struct X
{
  int  b;
  char a;
  char c;
};
```



Compound Types

Mental Model

What we write

```
struct Vector
{
  double length()
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector v { 1, 1 };
  std::cout << v.length() << std::endl;
}
```

Compound Types

Mental Model

What we write

```
struct Vector
{
  double length()
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector v { 1, 1 };
  std::cout << v.length() << std::endl;
}
```

A member function

Compound Types

Mental Model

What we write

```
struct Vector
{
  double length()
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector v { 1, 1 };
  std::cout << v.length() <<
}
```

How to call a member function

Compound Types

Mental Model

≈ What the compiler sees

```
struct Vector
{
  int x;
  int y;
};

double length(Vector* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector v { 1, 1 };
  std::cout << length(&v) << std::endl;
}
```

Compound Types

Mental Model

≈ What the compiler sees

```
struct Vector
{
  int x;
  int y;
```

≈ what the compiler translates member functions to

```
double length(Vector* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector v { 1, 1 };
  std::cout << length(&v) << std::endl;
}
```

Compound Types

Mental Model

≈ What the compiler sees

```
struct Vector
{
  int x;
  int y;
};

double length(Vector* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector v { 1, 1 };
  std::cout << length(&v) <<
}
```

How the compiler calls the member function

Compound Types

Mental Model

- We call member functions *on* objects
- The compiler translates member functions to *ordinary* functions which takes the object as the *first* parameter
- Then every call to a member function is just translated to a normal function call.
- This means that member functions are **NOT** stored in the object itself. So `length()` doesn't change the memory representation of `Vector` *at all*

Compound Types

`const` objects

```
struct Vector
{
  double length()
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector v { 1, 1 };
  std::cout << v.length() << std::endl;
}
```

Compound Types

`const` objects

```
struct Vector
{
  double length()
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector v { 1, 1 };
  std::cout << v.length() << std::endl;
}
```

Works!

LINKÖPING UNIVERSITY

Compound Types

const objects

```cpp
struct Vector
{
  double length()
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector const v { 1, 1 };
  std::cout << v.length() << std::endl;
}
```

Compound Types

`const` objects

```
struct Vector
{
  double length()
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector const v { 1, 1 };
  std::cout << v.length() << std::endl;
}
```

Compiler Error...

Compound Types

`const` objects

```
struct Vector
{
  double length()
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector const v { 1, 1 };
  std::cout << v.length() << std::endl;
}
```

Why?

Compound Types

Let’s translate to our mental model

Compound Types

Mental Model

```
struct Vector
{
  int x;
  int y;
};

double length(Vector* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector const v { 1, 1 };
  std::cout << length(&v) << std::endl;
}
```

This is what the compiler sees

Compound Types

Mental Model

```
struct Vector
{
  int x;
  int y;
};

double length(Vector* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector const v { 1, 1 };
  std::cout << length(&v) <<
}
```

This is what the compiler sees

What is the type of &v?

Compound Types

Mental Model

```
struct Vector
{
  int x;
  int y;
};

double length(Vector* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector const v { 1, 1 };
  std::cout << length(&v) <<
}
```

This is what the compiler sees

It is `Vector const*`

Compound Types

Mental Model

```
struct Vector
{
  int x;
  int y;
};

double length(Vector* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector const v { 1, 1 };
  std::cout << length(&v) <<
}
```

This is what the compiler sees

Which doesn’t match the parameter...

LINKÖPING UNIVERSITY

Compound Types

Mental Model

```
struct Vector
{
  int x;
  int y;
};

double length(Vector* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector const v { 1, 1 };
  std::cout << length(&v) << std::endl;
}
```

This is what the compiler sees

We need the parameter to take `Vector const*`

Compound Types

Enter `const` member functions!

The code

```
struct Vector
{
  double length() const
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector const v { 1, 1 };
  cout << v.length() << endl;
}
```

Compound Types

Enter `const` member functions!

The code

```
struct Vector
{
  double length() const
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector const v { 1, 1 };
  cout << v.length() << endl;
}
```

The compilers view

```
struct Vector
{
  int x;
  int y;
};

double length(Vector const* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector const v { 1, 1 };
  cout << length(&v) << endl;
}
```

Compound Types

Enter `const` member functions!

The code

```
struct Vector
{
  double length() const
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector const v { 1, 1 };
  cout << v.length() << endl;
}
```

The compilers view

```
struct Vector
{
  int x;
  int y;
};

double length(Vector const* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector const v { 1, 1 };
  cout << length(&v) << endl;
}
```

Compound Types

Enter `const` member functions!

The code

```
struct Vector
{
  double length() const
  {
    double x2 { x * x };
    double y2 { y * y };
    return std::sqrt(x2 + y2);
  }

  int x;
  int y;
};

int main()
{
  Vector const v { 1, 1 };
  cout << v.length() << endl;
}
```

Works!

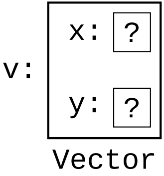
The compilers view

```
struct Vector
{
  int x;
  int y;
};

double length(Vector const* this)
{
  double x2 { this->x * this->x };
  double y2 { this->y * this->y };
  return std::sqrt(x2 + y2);
}

int main()
{
  Vector const v { 1, 1 };
  cout << length(&v) << endl;
}
```

Compound Types

Initialization

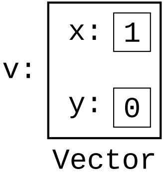
```
struct Vector
{
  int x;
  int y;
};

int main()
{
  Vector v { };
}
```



Compound Types

Initialization

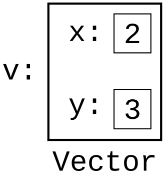
```
struct Vector
{
  int x { 1 };
  int y { 0 };
};

int main()
{
  Vector v { };
}
```



Compound Types

Initialization

```
struct Vector
{
  int x { 1 };
  int y { 0 };
};

int main()
{
  Vector v { 2, 3 };
}
```



Compound Types

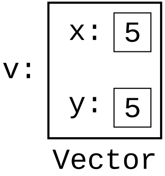
Initialization

- If we don't explicitly initialize the data members they will be undefined (in the first example)
- But we can give each data member a *default* value by adding initialization to the data members (second example)
- But we can always override the default if we explicitly initialize the data members (third example)

Compound Types

Constructor

```
struct Vector
{
  Vector(int value)
    : x { value }, y { value }
  {
  }

  int x;
  int y;
};

int main()
{
  Vector v { 5 };
}
```

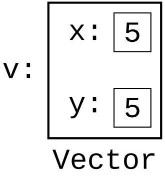


Compound Types

Constructor

```
struct Vector
{
  Vector(int value)
    : x { value }, y { value }
  {
  }

  int x;
  int y;
};

int main()
{
  Vector v { 5 };
}
```

Constructor

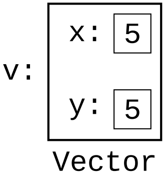


Compound Types

Constructor

```cpp
struct Vector
{
  Vector(int value)
    : x { value }, y { value }
  {
  }

  int x;
  int y;
};

int main()
{
  Vector v { 5 };
}
```

Constructor call

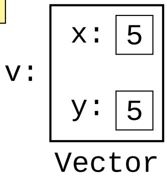


Compound Types

Constructor

```
struct Vector
{
  Vector(int value)
    : x { value }, y { value }
  {
  }

  int x;
  int y;
};

int main()
{
  Vector v { 5 };
}
```

member initializer list



Compound Types

member initialializer list

- The *member initializer list* is a special syntax for constructors
- It allows us to *override* the default initializers for data members in a specific constructor call
- The member initializer list is a comma separated list of initialization statements for all/any data members (see example on previous slide)
- This is prefered over assignment (see next example)

Compound Types

Member initializer list vs. assignment

```
class X
{
public:
  X(int c)
  {
    a = c;
    b = c + 1;
  }
private:
  int a;
  int b;
};
```

Don't write code like this...

Compound Types

Member initializer list vs. assignment

```
class X
{
public:
  X(int c)
  {
    a = c;
    b = c + 1;
  }
private:
  int const a;
  int b;
};
```

...It doesn't work for const

Compound Types

Member initializer list vs. assignment

```
class X
{
public:
  X(int c)
  {
    a = c;
    b = c + 1;
  }
private:
  int const a;
  int b;
};
```

...It doesn’t work for const

Compound Types

Member initializer list vs. assignment

```
class X
{
public:
  X(int c)
  {
    a = c;
    b = c + 1;
  }
private:
  int const a;
  int b;
};
```

...It doesn't work for const

Compound Types

Member initializer list vs. assignment

```
class X
{
public:
  X(int c)
    : a { c },
      b { c + 1 }
  {
  }
private:
  int a;
  int b;
};
```

Prefer this...

Compound Types

Member initializer list vs. assignment

```
class X
{
public:
  X(int c)
    : a { c },
      b { c + 1 }
  {
  }
private:
  int const a;
  int b;
};
```

... It *does* work for const!

Compound Types

What will be printed?

```
class X
{
public:
  void print(int&)              { std::cout << "1"; }
  void print(int const&)        { std::cout << "2"; }
  void print(int const&) const  { std::cout << "3"; }
};

int main()
{
  X x1 { };
  X const x2 { };
  int y1 { };
  int const y2 { };

  x1.print(y1);
  x2.print(y1);
  x1.print(y2);
  x2.print(y2);
}
```

LINKÖPING
UNIVERSITY

Operator Overloading

Introduction

- A powerful aspect of C++ is the fact that we can define operators for our own user-defined types
- This allows us to greatly simplify how we *use* our classes/structs (i.e. simplify the interface)
- This is called *operator overloading*
- If used correctly it will make our code easier to understand by relating it to mathmatical notation
- **BUT**, if used *incorrectly* it will make our code *harder* to understand, so we have to be careful...

Operator Overloading

Extending `Vector`

```
Vector v { 1, 2 };
Vector u { 3, 1 };

// This is our aim
Vector w { 3*v + u };

assert(w.x == 3*v.x + u.x);
assert(w.y == 3*v.y + u.y);
```

Operator Overloading

How it works

```
3*v + u
```

Operator Overloading

How it works

```
(3*v) + u
```

Operator Overloading

How it works

```
((3*v) + u)
```

Operator Overloading

How it works

```
operator+((3*v), u)
```

Operator Overloading

How it works

```
operator+(operator*(3, v), u)
```

Operator Overloading

How it works

- Whenever the compiler encounters an operator involving a class type it knows that this must be an operator overload
- If it for example finds `a+b` then the compiler will translate it to a *function call*
- Specifically, the compiler will call: `operator+(a, b)`
- Note that `a` is to the left of `+` so it will be the first parameter and `b` is to the right so it is the second parameter.
- If `operator+(a, b)` doesn't exist, then it will instead try `a.operator+(b)`
- **Note:** If both versions exists then it is ambigious...
- Read more: https://en.cppreference.com/w/cpp/language/operators

Operator Overloading

When it *works*

```
// With operator overloads
5*(u + v) + w;

// Without
add(multiply(5, add(u, v)), w);
```

Operator Overloading

When it *works*

```
// With operator overloads
5*(u + v) + w;

// Without
add(multiply(5, add(u, v)), w);
```

Which is easier to understand/read?

Operator Overloading

When it *doesn't* work...

u * v

Operator Overloading

When it *doesn’t* work...

u * v
Dot product?

Operator Overloading

When it *doesn't* work...

u * v

Dot product?

Scalar product?

Operator Overloading

When it *doesn’t* work...

u * v

Dot product?

Scalar product?

Element-wise multiplication?

Operator Overloading

When it *doesn't* work...

- **Lesson #1**: Operator overloading only works if it is *obvious* what it means.
- The example given on the previous slide multiplies a vector with a vector
- But there are multiple ways to define "vector multiplication" so it is not clear from just reading the code what is meant.
- This is **bad**, but accepted by the language.
- It is our job to *carefully* consider whether an operator overload will lead to ambiguity or not...

Operator Overloading

When it *doesn't* work...

```
Vector v { 1, 2 };
Vector u { 3, 1 };
Vector w { v + u };

// What do we expect to be printed?
cout << v.x << endl;
```

Operator Overloading

When it *doesn’t* work...

Compare with the `int` case

Operator Overloading

When it *doesn't* work...

```
int v { 1 };
int u { 3 };
int w { v + u };

// Here we expect v to be unchanged
cout << v << endl;
```

Operator Overloading

When it *doesn't* work...

```
Vector v { 1, 2 };
Vector u { 3, 1 };
Vector w { v + u };

// So here v.x should be unchanged
cout << v.x << endl;
```

Operator Overloading

When it *doesn't* work...

- **Lesson #2:** Operators should have the *expected* behaviour
- This means that an operators semantics should be as similar to the behaviour of corresponding operator on fundamental types
- On the previous slide we for example saw that `operator+` should *not* modify any of the operands.
- So before doing an operator overload, ask yourself whether it behaves the same way as for the builtin types.
- **Note:** It is *legal* to break the semantics, but it is a **very** bad practice to do so.

Operator Overloading

Design principle

When overloading an operator make sure that:

Operator Overloading

Design principle

When overloading an operator make sure that:

- The behaviour is obvious and makes sense

Operator Overloading

Design principle

When overloading an operator make sure that:

- The behaviour is obvious and makes sense
- It is similar to the fundamental type operators

LINKÖPING UNIVERSITY

Value categories

Assignments

```
int x { 3 };
x = 5;      // OK
3 = 5;      // NOT OK
x + 1 = 3; // NOT OK
```

Value categories

Assignments

```
int x { 3 };
x = 5;      // OK
3 = 5;      // NOT OK
x + 1 = 3; // NOT OK
```

... Why?

Value categories

Assignments

- `x` is what is called an *lvalue*
- *lvalues* are expressions that refer to a specific *object/variable*
- Whenever we use the expression `x` in a scope it will always refer to the *same* object
- expressions such as `3`, `int{}` and `x+1` are *rvalues*
- *rvalues* are expressions that generate a new *value* whenever it appears.

Value categories

Assignments

- Another way to differentiate between them is to think about assignments (Note that these intutions aren't always correct).
- `x` is an *lvalue* (**l**eft-hand-side **value**) if it *can* appear on left side of an assignment.
- `x+1` is an *rvalue* (**r**ight-hand-side **value**) since it can *only* appear on the right-hand-side of an assignment.

Value categories

Assignments

- If an object have *identity*, i.e. if there is a way for us to *refer* to the object. Then every expression that refers to that object will be an *lvalue*.
- For example: if there is a pointer to the object, if the object is a variable or if it is a part of a bigger object (like an array or a class).
- So things like: `*ptr`, `array[0]` etc. are also *lvalues*.
- *rvalues* are generally expressions that are *not lvalues*.

Value categories

lvalues & rvalues

lvalues

```
x
*ptr
array[0]
// etc.
```

rvalues

```
5
int{}
x + 1
// etc.
```

Value categories

What is the value category of the expression?

```
int const x { };
int zero()
{
  return x;
}

zero() // <- what is the value category?
```

Value categories

What is the value category of the expression?

```
int array[3];

*(&array[0] + 1) // <- what is the value category?
```

Value categories

What is the value category of the expression?

```
int const x { };
int& zero()
{
  return x;
}

zero() // <- what is the value category?
```

www.liu.se

LINKÖPING UNIVERSITY