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

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Pointers & References

Types of indirection

- Data pointers
- Function pointers
- References

Pointers & References

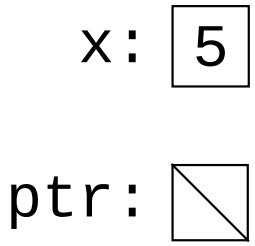
Types of indirection

- **Data pointers**
- Function pointers
- References

Pointers & References

Data pointer

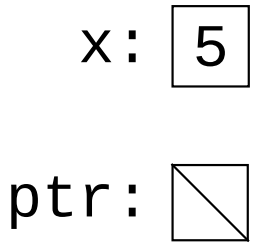
```
1 int x    { 5 };
2 int* ptr { nullptr };
3
4 ptr = &x;
5 *ptr = 7;
6
7 std::cout << x << std::endl;
```



Pointers & References

Data pointer

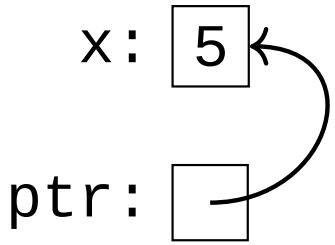
```
1 int x    { 5 };
2 int* ptr { nullptr };
3
4 ptr = &x;
5 *ptr = 7;
6
7 std::cout << x << std::endl;
```



Pointers & References

Data pointer

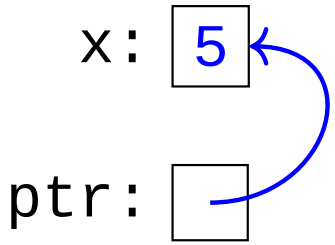
```
int x    { 5 };
int* ptr { nullptr };

ptr = &x;
*ptr = 7;

std::cout << x << std::endl;
```



Pointers & References

Data pointer

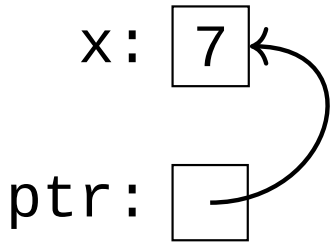
```
1 int x    { 5 };
2 int* ptr { nullptr };
3
4 ptr = &x;
5 *ptr = 7;
6
7 std::cout << x << std::endl;
```



Pointers & References

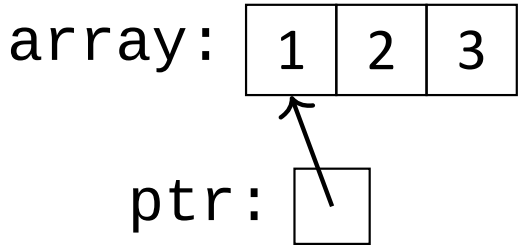
Data pointer

```
1 int x    { 5 };
2 int* ptr { nullptr };
3
4 ptr = &x;
5 *ptr = 7;
6
7 std::cout << x << std::endl;
```



Pointers & References

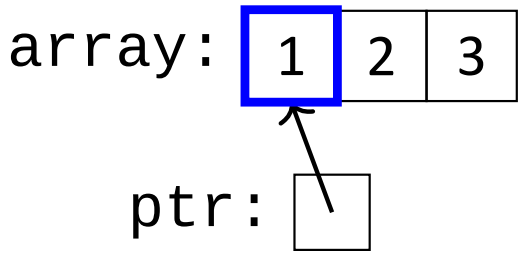
Pointers to arrays




```
1 int array[3] { 1, 2, 3 };
2 int* ptr { &array[0] };
```

Pointers & References

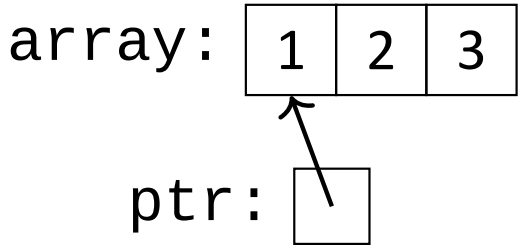
Pointers to arrays




```
1 int array[3] { 1, 2, 3 };
2 int* ptr { &array[0] };
```

Pointers & References

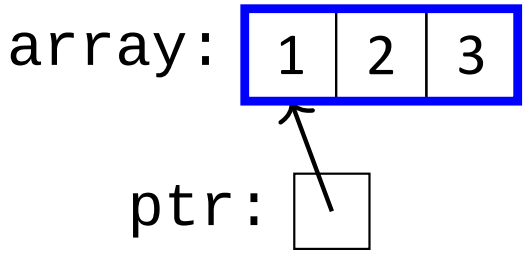
Pointers to arrays




```
1 int array[3] { 1, 2, 3 };
2 int (*ptr)[3] { &array };
```

Pointers & References

Pointers to arrays




```
1 int array[3] { 1, 2, 3 };
2 int (*ptr)[3] { &array };
```

Pointers & References

Arrays and pointers: What's the difference?

```
int (*array)[3]
```

```
int *array[3]
```

Pointers & References

Types of indirection

- Data pointers
- Function pointers
- References

Pointers & References

Types of indirection

- Data pointers
- **Function pointers**
- References

Pointers & References

Function pointers

```
int add(int x, int y){ /* ... */ }

int sub(int x, int y){ /* ... */ }

int main()
{
  int (*ptr)(int, int){ };

  ptr = &add;
  cout << (*ptr)(3, 2) << endl;

  ptr = &sub;
  cout << (*ptr)(3, 2) << endl;
}
```

Pointers & References

Function pointers

```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```

ptr: 

Pointers & References

Function pointers

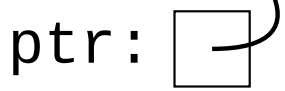
```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```

ptr: 

Pointers & References

Function pointers

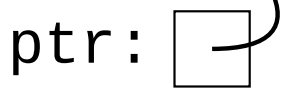
```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```



Pointers & References

Function pointers

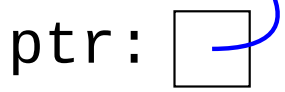
```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```



Pointers & References

Function pointers

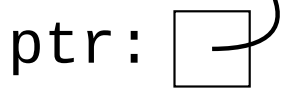
```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```



Pointers & References

Function pointers

```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```



Pointers & References

Function pointers

```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```

Pointers & References

Function pointers

```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```

Pointers & References

Function pointers

```
1  int add(int x, int y){ /* ... */ }
2
3  int sub(int x, int y){ /* ... */ }
4
5  int main()
6  {
7    int (*ptr)(int, int){ };
8
9    ptr = &add;
10   cout << (*ptr)(3, 2) << endl;
11
12   ptr = &sub;
13   cout << (*ptr)(3, 2) << endl;
14 }
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

How to read these “special” pointers

```
int (*(*ptr)(int))[5]
```

Pointers & References

(confusing) Example

```
int array[2] { };

int (*fun(int x, int y))[2]
{
    array[0] = x;
    array[1] = y;
    return &array;
}

int main()
{
    int (*a)[2] { fun(1, 2) };
    cout << (*a)[0] + (*a)[1] << endl;
}
```

Pointers & References

(better) Example

```
int array[2] { };
using array_ptr = int(*)[2];

array_ptr fun(int x, int y)
{
    array[0] = x;
    array[1] = y;
    return &array;
}

int main()
{
    array_ptr a { fun(1, 2) };
    cout << (*a)[0] + (*a)[1] << endl;
}
```

Pointers & References

Types of indirection

- Data pointers
- Function pointers
- References

Pointers & References

Types of indirection

- Data pointers
- Function pointers
- **References**

Pointers & References

References (or variable aliases)

```
int x { 5 }; // normal variable
int& y { x }; // lvalue-reference
int const& z { y }; // const lvalue-reference

x = 3;
assert(x == 3 && x == y && y == z);

y = 7;
assert(y == 7 && x == y && y == z);

z = 2; // NOT OK
```

Pointers & References

Why?

```
1  void increase(int a)
2  {
3      ++a;
4  }
5
6  int main()
7  {
8    int x { 0 };
9    increase(x);
10   cout << x << endl; // prints 0
11 }
```

Pointers & References

Why?

```
1  void increase(int& a)
2  {
3      ++a;
4  }
5
6  int main()
7  {
8    int x { 0 };
9    increase(x);
10   cout << x << endl; // prints 1
11 }
```

Pointers & References

What type of entity is x?

```
int *(*x())[3]
```

Pointers & References

What type of entity is x?

```
int (*x[3])()
```

LINKÖPING UNIVERSITY

Value categories

Assignments

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

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?
```

LINKÖPING UNIVERSITY

Class Types

All class types

- `struct`
- `class`
- `union` (later)

Class Types

Classes and structs are the same thing!

```
struct Vector_Struct
{

  int x;
  int y;
};
```

```
class Vector_Class
{

  int x:
  int y;
};
```

Class Types

Classes and structs are the same thing!

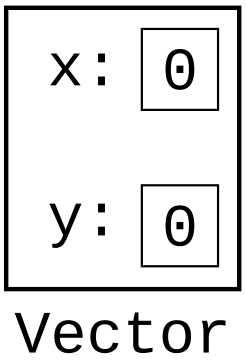
```
1 struct Vector_Struct
2 {
3 public:
4   int x;
5   int y;
6 };
```

```
1 class Vector_Class
2 {
3 private:
4   int x:
5   int y;
6 };
```

Class Types

Mental Model

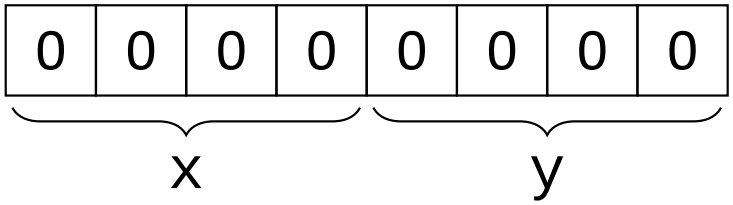
```
1 struct Vector
2 {
3   int x { 0 };
4   int y { 0 };
5 };
```



Class Types

Mental Model

```
1 struct Vector
2 {
3   int x { 0 };
4   int y { 0 };
5 };
```



Class Types

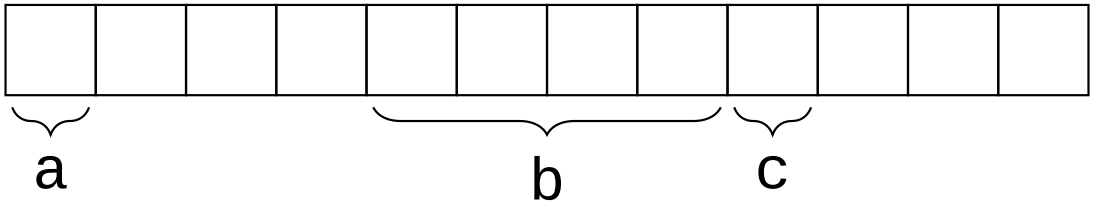
Padding & Alignment

```
1 struct X
2 {
3   char a;
4   int  b;
5   char c;
6 };
```

Class Types

Padding & Alignment

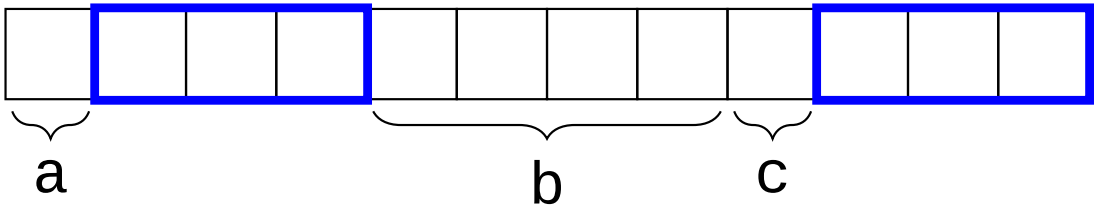
```
1 struct X
2 {
3   char a;
4   int  b;
5   char c;
6 };
```



Class Types

Padding & Alignment

```
1 struct X
2 {
3   char a;
4   int  b;
5   char c;
6 };
```



Class Types

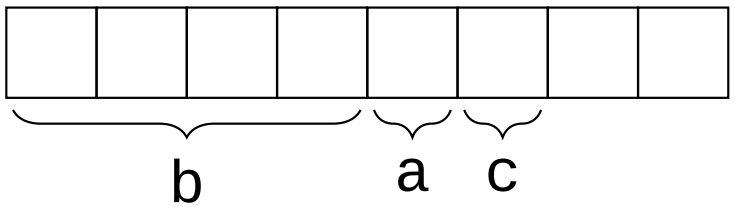
Padding & Alignment

```
1 struct X
2 {
3   int  b;
4   char a;
5   char c;
6 };
```

Class Types

Padding & Alignment

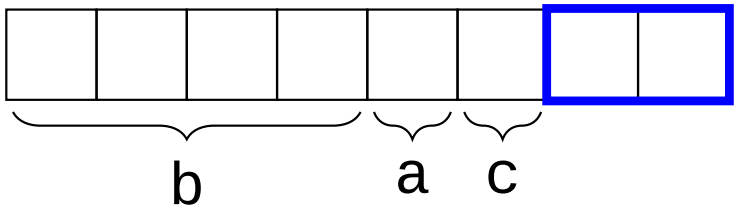
```
1 struct X
2 {
3   int  b;
4   char a;
5   char c;
6 };
```



Class Types

Padding & Alignment

```
1 struct X
2 {
3   int  b;
4   char a;
5   char c;
6 };
```



Class Types

Mental Model

```
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;
}
```

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

Class 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;
}
```

Class 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!

Class 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;
}
```

Class 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...

Class 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?

Class 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;
}
```

Class Types

Enter `const` member functions!

```
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;
}
```

Class Types

Enter `const` member functions!

```
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;
}
```

```
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;
}
```

Class Types

Enter `const` member functions!

```
1  struct Vector
2  {
3    double length() const
4    {
5      double x2 { x * x };
6      double y2 { y * y };
7      return std::sqrt(x2 + y2);
8    }
9
10   int x;
11   int y;
12 };
13
14 int main()
15 {
16   Vector const v { 1, 1 };
17   cout << v.length() << endl;
18 }
```

```
1  struct Vector
2  {
3    int x;
4    int y;
5  };
6
7  double length(Vector const* this)
8  {
9    double x2 { this->x * this->x };
10   double y2 { this->y * this->y };
11   return std::sqrt(x2 + y2);
12 }
13
14 int main()
15 {
16   Vector const v { 1, 1 };
17   cout << length(&v) << endl;
18 }
```

Class Types

Enter `const` member functions!

```
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;
}
```

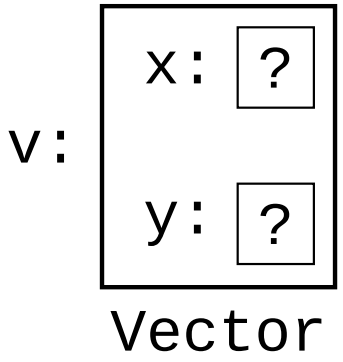
Works!

```
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;
}
```

Class Types

Initialization

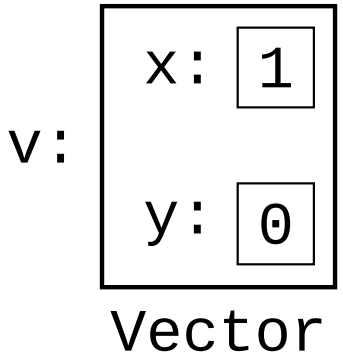
```
1  struct Vector
2  {
3    int x;
4    int y;
5  };
6
7  int main()
8  {
9    Vector v { };
10 }
```



Class Types

Initialization

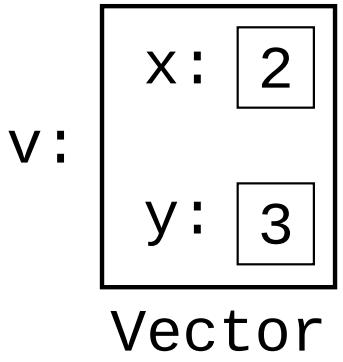
```
1  struct Vector
2  {
3    int x { 1 };
4    int y { 0 };
5  };
6
7  int main()
8  {
9    Vector v { };
10 }
```



Class Types

Initialization

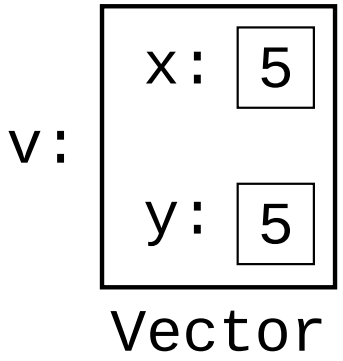
```
1  struct Vector
2  {
3    int x { 1 };
4    int y { 0 };
5  };
6
7  int main()
8  {
9    Vector v { 2, 3 };
10 }
```



Class Types

Constructor

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

  int x;
  int y;
};

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

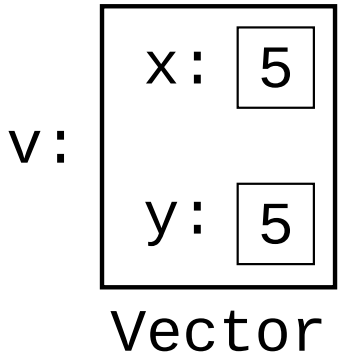


Class Types

Constructor

```
1  struct Vector
2  {
3    Vector(int value)
4      : x { value }, y { value }
5    {
6    }
7
8    int x;
9    int y;
10 };
11
12 int main()
13 {
14   Vector v { 5 };
15 }
```

Constructor

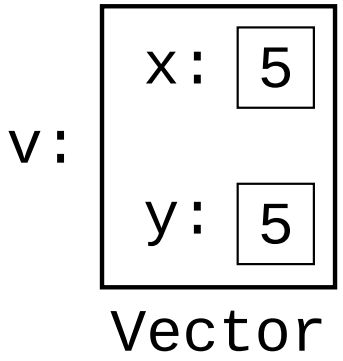


Class Types

Constructor

```
1  struct Vector
2  {
3    Vector(int value)
4      : x { value }, y { value }
5    {
6    }
7
8    int x;
9    int y;
10 };
11
12 int main()
13 {
14   Vector v { 5 };
15 }
```

Constructor call

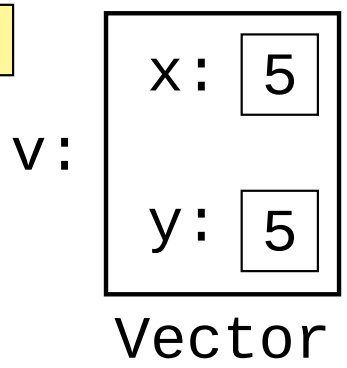


Class Types

Constructor

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

  int x;
  int y;
};

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

member initializer list



Class Types

Member initializer list vs. assignment

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

private:
  int a;
  int b;
};
```

Class Types

Member initializer list vs. assignment

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

private:
  int const a;
  int b;
};
```

Class Types

Member initializer list vs. assignment

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

private:
  int const a;
  int b;
};
```

Class Types

Member initializer list vs. assignment

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

private:
  int const a;
  int b;
};
```

Class Types

Member initializer list vs. assignment

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

private:
  int a;
  int b;
};
```

Class Types

Member initializer list vs. assignment

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

private:
  int const a;
  int b;
};
```

Class 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

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

When it *works*

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

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

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...

```
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

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

User-defined conversions

Type conversions

```
class Cls
{
public:
  Cls(int i) : i{i} { }
  operator int() const
  {
    return i;
  }
private:
  int i;
};
```

User-defined conversions

Explicit keyword

```
class Cls
{
public:
  explicit Cls(int i) : i{i} { }
  explicit operator int() const
  {
    return i;
  }
private:
  int i;
};
```

User-defined conversions

Contextual Conversion

```
1  struct Cls
2  {
3    explicit operator bool() const { return flag; }
4    bool flag{};
5  };
6  int main()
7  {
8    Cls c{};
9    if (c)
10   {
11     // ...
12   }
13 }
```

www.liu.se

LINKÖPING
UNIVERSITY