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

Fundamentals II

Christoffer Holm

Department of Computer and information science

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

- Variable which stores memory addresses
- Knows what type of data is located at the other end
- Has a special value called `nullptr`
- This special value indicates that the pointers points to nothing
- Is associated with specific operators: *dereference* (`*`) and *address-of* (&)

Pointers & References

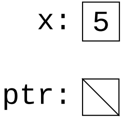
Data pointer

- The dereference operator takes a *pointer* and returns the data it points to
- The address-of operator takes a variable/object and returns a *pointer* to that object

Pointers & References

Data pointer

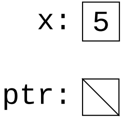
```
int x    { 5 };
int* ptr { nullptr };

ptr = &x;
*ptr = 7;

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



Pointers & References

Data pointer

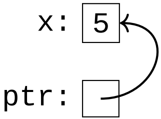
```
int x     { 5 };
int* ptr { nullptr };

ptr = &x;
*ptr = 7;

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



Pointers & References

Data pointer

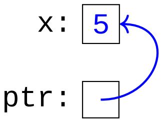
```
int x    { 5 };
int* ptr { nullptr };

ptr = &x;
*ptr = 7;

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



Pointers & References

Data pointer

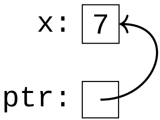
```
int x    { 5 };
int* ptr { nullptr };

ptr = &x;
*ptr = 7;

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



Pointers & References

Data pointer

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

ptr = &x;
*ptr = 7;

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



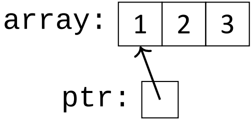
Pointers & References

Pointers to arrays

- In C++ we can have pointers to specific elements in the array
- This means we can represent the array as a pointer to the first element (but then we have to manually keep track of the number of elements)
- But we can also have pointers to the *whole* array
- These have the advantage that they automatically remember the size of the array

Pointers & References

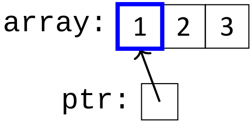
Pointers to arrays




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

Pointers & References

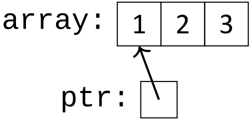
Pointers to arrays




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

Pointers & References

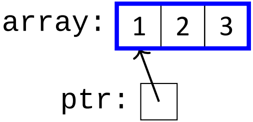
Pointers to arrays




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

Pointers & References

Pointers to arrays




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

Pointers & References

Pointers to arrays

- The first example show a pointer to an element
- The second example is a pointer to the whole array
- Pointers to specific element have type: `int*`
- pointer to an array has type: `int (*ptr)[3]`
- Compare with: `int* ptr[3]`, what does this mean?

Pointers & References

Arrays and pointers: What's the difference?

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

A pointer to an array of 3 `int` elements

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

An array of 3 `int*` elements

Pointers & References

Types of indirection

- Data pointers
- Function pointers
- References

Pointers & References

Types of indirection

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

Pointers & References

Function pointers

- Also contains a memory address, but this time it points to *executable code* (specifically a function) rather than data
- Knows the signature of the function
- Uses the dereference and adress-of operators (but on functions instead of data)
- It also has the *function call* operator which allows us to *call* the function it points to

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

```
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;
}
```

ptr:

Pointers & References

Function pointers

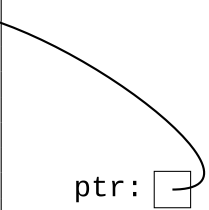
```
int add(int x, int y){ /* ... */ }

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;
}
```

ptr: 

Pointers & References

Function pointers

```cpp
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

```cpp
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

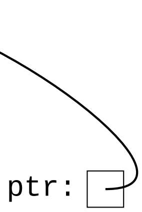
```
int add(int x, int y){ /* ... */ }

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;
}
```

ptr:

Pointers & References

Function pointers

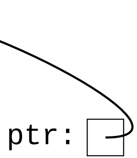
```
int add(int x, int y){ /* ... */ }

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

```
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

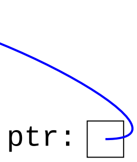
```cpp
int add(int x, int y){ /* ... */ }

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;
}
```

ptr:

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

How to read these “special” pointers

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

Pointers & References

How to read these “special” pointers

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

`ptr` is

Pointers & References

How to read these “special” pointers

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

`ptr` is

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer to a function taking (`int`)

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer to a function taking `(int)`

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer to a function taking `(int)`, which returns a pointer

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer to a function taking `(int)`, which returns a pointer

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer to a function taking `(int)`, which returns a pointer to an array of size 5

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer to a function taking `(int)`, which returns a pointer to an array of size 5

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer to a function taking `(int)`, which returns a pointer to an array of size 5 containing `int` elements.

Pointers & References

How to read these “special” pointers

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

`ptr` is a pointer to a function taking `(int)`, which returns a pointer to an array of size 5 containing `int` elements.

Pointers & References

The spiral rule

1. Start from the unknown name
2. If there are parenthesis with parameters to the right, then it is a function.
3. If there are brackets with a size to the right, then it is an array.
4. Otherwise it is a variable.
5. Read to the left until reaching the beginning *or* until an open parenthesis.
6. If we reached the end then we are done.
7. Otherwise: Jump to the matching closing parenthesis.
8. Read to the right.
9. If we find open parenthesis then it is a function.
10. If we find square bracket then it is an array.
11. Go back to the previously found open parenthesis and goto step 5.

Pointers & References

Another way to view it

To figure out the declaration, look at how you would use it:

- `*ptr` gives us `int` ⇒ `int *ptr`
- `(*ptr)[0]` gives us `int` ⇒ `int (*ptr)[3]`
- `(*ptr)(1, 2)` gives us `void` ⇒ `void (*ptr)(int, int)`
- `(*ptr[0])()` gives us `int` ⇒ `int (*ptr[3])()`
- etc.

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

Example

- These types of declarations are generally very hard to grasp
- It is not always clear what is actually defined
- Because of this it is *highly* recommended to abstract these away using a *type alias* (`using`)

Pointers & References

Example

- The statement: `using number = int;` creates a *type alias* for `int` which we call `number`
- What this means more concretely is that we create an alternate *name* for the type `int` (namely `number`)
- There are many type aliases in the language which can be used to make the code easier to modify and understand.
- An example is `std::size_t` which is the smallest type needed to index *all* bytes in memory. On a 32-bit systems this might be an alias for `std::uint32_t` (which itself is an alias representing an unsigned integer of size 32 bits).

Pointers & References

Example

- In this example we create an alias `array_ptr` which represents the type `int(*)[2]` (an array pointer without a name, compare with `int(*array)[2]`)
- By doing this we can use `array_ptr` as a “normal” type without having to deal with the nested parenthesis.
- This technique will usually make things *way* easier to read.

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)

```cpp
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

References

- In the example on the previous slide `x`, `y` and `z` all refer to the *same* variable.
- So if we change `x` this will be reflected in `y` and `z` (even though it is const). Likewise if we modify `y`.
- So just because `z` is `const` that doesn't *necessarily* mean that its value won't change.
- Instead it just means that we are not allowed to modify the value *through* `z`.

Pointers & References

Why?

```cpp
void increase(int a)
{
    ++a;
}

int main()
{
  int x { 0 };
  increase(x);
  cout << x << endl; // prints 0
}
```

Pointers & References

Why?

```cpp
void increase(int& a)
{
    ++a;
}

int main()
{
  int x { 0 };
  increase(x);
  cout << x << endl; // prints 1
}
```

Pointers & References

Why?

- References are useful in combination with functions
- This allows us to have *in-out* parameters.
- I.e. parameters which changes the variable we passed in
- If we don't pass parameters as references we just get a copy of the variable local to the function.
- But if we take the parameter `a` as a reference, then we get an *alias* to the original variable `x`.

Pointers & References

What type of entity is x?

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

Pointers & References

What type of entity is x?

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

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

Since C++11 value categories have evolved, but more on that next week

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;
};
```

What is the difference?

Class Types

Classes and structs are the same thing!

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

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

Class Types

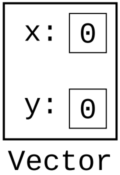
`struct` vs. `class`

- 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 next week)
- Besides this they are *functionally* the same thing

Class Types

Mental Model

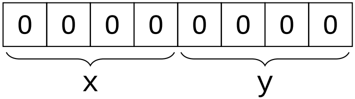
```
struct Vector
{
  int x { 0 };
  int y { 0 };
};
```



Class Types

Mental Model

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



Class 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)

Class 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.

Class 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*

Class 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*)

Class Types

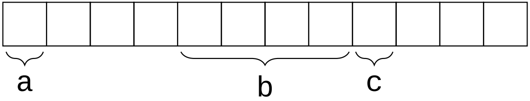
Padding & Alignment

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

Class Types

Padding & Alignment

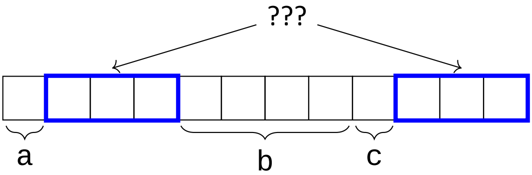
```
struct X
{
  char a;
  int  b;
  char c;
};
```



Class Types

Padding & Alignment

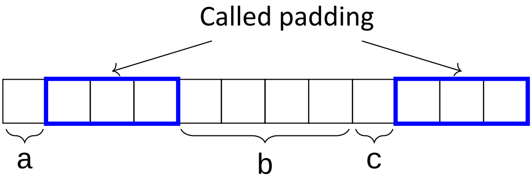
```
struct X
{
  char a;
  int  b;
  char c;
};
```



Class Types

Padding & Alignment

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



Class 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

Class 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)

Class 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)

Class Types

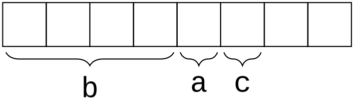
Padding & Alignment

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

Class Types

Padding & Alignment

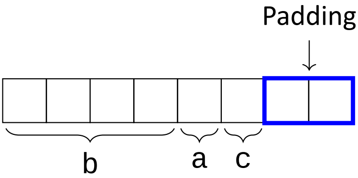
```
struct X
{
  int  b;
  char a;
  char c;
};
```



Class Types

Padding & Alignment

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



Class 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;
}
```

Class Types

What we write

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;
}
```

A member function

Class 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

Class 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;
}
```

Class 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;
}
```

Class 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

Class 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*

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

Let’s translate to our mental model

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;
}
```

This is what the compiler sees

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) <<
}
```

This is what the compiler sees

What is the type of &v?

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) <<
}
```

This is what the compiler sees

It is `Vector const*`

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;
}
```

This is what the compiler sees

Which doesn’t match the parameter...

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) <<
}
```

This is what the compiler sees

We need the parameter to take `Vector const*`

Class 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;
}
```

Class 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;
}
```

Class 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;
}
```

Class 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!

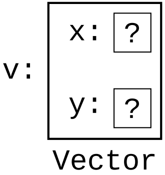
The compilers view

```
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;
}
```

Class Types

Initialization

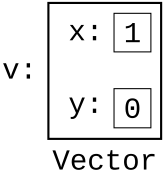
```
struct Vector
{
  int x;
  int y;
};

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



Class Types

Initialization

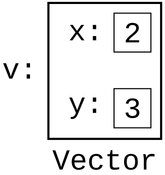
```
struct Vector
{
  int x { 1 };
  int y { 0 };
};

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



Class Types

Initialization

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

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



Class Types

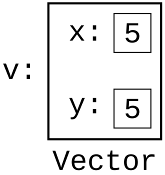
Initialization

- 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)

Class Types

Constructor

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

  int x;
  int y;
};

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

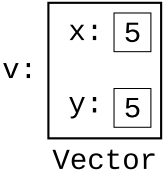


Class Types

Constructor

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

  int x;
  int y;
};

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

Constructor

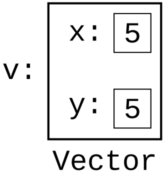


Class Types

Constructor

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

  int x;
  int y;
};

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

Constructor call

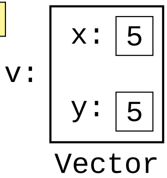


Class Types

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

- 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)

Class 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...

Class 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

Class 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

Class 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

Class Types

Member initializer list vs. assignment

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

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

Prefer this...

Class 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!

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

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

User-defined conversions

Type conversions

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

User-defined conversions

Type conversions

- A constructor that can take **one** argument is called a *type converting constructor*;
- these constructors can be used by the compiler to perform conversions.
- The special operator `Cls::operator TYPE()` is called whenever the class `Cls` is converted to `TYPE`;
- the compiler is allowed to use this operator to perform implicit type conversions;
- but can also be explicitly called through casting.

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

Explicit keyword

- Declaring type converting constructors or operators as `explicit` means;
- the compiler is **not** allowed to use these functions for implicit type conversion;
- with the exception of `operator bool` which can be used for *contextual conversion*.

User-defined conversions

Contextual Conversion

```
struct Cls
{
  explicit operator bool() const { return flag; }
  bool flag{};
};
int main()
{
  Cls c{};
  if (c)
  {
    // ...
  }
}
```

User-defined conversions

Read more

- https://en.cppreference.com/w/cpp/language/converting_constructor
- https://en.cppreference.com/w/cpp/language/cast_operator

www.liu.se