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

Christoffer Holm

Department of Computer and information science

1 Data types
2 Functions
3 Conversions
4 Initialization

# Data types

Data type categories

- Fundamental types
- Array types
- Enum types
- Class types (later)
- Pointer/Reference types (later)

# Data types

Data type categories

- **Fundamental types**
- Array types
- Enum types
- Class types (later)
- Pointer/Reference types (later)

# Data types

Fundamental data types

- Integer types
- Character types
- Floating-point types
- Other types

# Data types

Fundamental data types

- **Integer types**
- Character types
- Floating-point types
- Other types

# Data types

Integer types

- **Basic Type:** `int` - At least 16 bits, but usually 32 bits
- **Size Modifiers:**
  - `short` - Smallest possible (at least 16 bits)
  - `long` - Guarantee at least 32 bits
  - `long long` - Largest possible (at least 64 bits)
- **Signedness Modifiers:**
  - `signed` - Can represent positive and negative values
  - `unsigned` - Can only represent positive values

# Data types

Integer types

- Each integer type is constructed by combining a *size* and a *signedness* modifier.
- The order of the keywords can be anything, but it is usually: `<signedness> <size> int`
- For example: `unsigned long long int`
- `int` is commonly omitted if a modifier is used, for example: `unsigned short` instead of `unsigned short int`
- The `signed` modifier is usually omitted since it is the default for all types. So for example: `int` instead of `signed int`.

# Data types

## Integer types

| Type | Size[1] | Value Range[1] |
|---|---|---|
| `short` | 16 bits | $[-2^{15}, 2^{15}-1]$ |
| `int` | 32 bits | $[-2^{31}, 2^{31}-1]$ |
| `long` | 32 bits | $[-2^{31}, 2^{31}-1]$ |
| `long long` | 64 bits | $[-2^{63}, 2^{63}-1]$ |
| | | |

[1] Very common values, but not guaranteed on all systems.

# Data types

## Integer types

| Type | Size[1] | Value Range[1] |
|---|---|---|
| `short` | 16 bits | $[-2^{15}, 2^{15}-1]$ |
| `int` | 32 bits | $[-2^{31}, 2^{31}-1]$ |
| `long` | 32 bits | $[-2^{31}, 2^{31}-1]$ |
| `long long` | 64 bits | $[-2^{63}, 2^{63}-1]$ |
| `unsigned short` | 16 bits | $[0, 2^{16}-1]$ |
| `unsigned int` | 32 bits | $[0, 2^{32}-1]$ |
| `unsigned long` | 32 bits | $[0, 2^{32}-1]$ |
| `unsigned long long` | 64 bits | $[0, 2^{64}-1]$ |

[1] Very common values, but not guaranteed on all systems.

# Data types

Integer types

- Since the `signed` and `unsigned` values are stored using the same number of bits it means that they can represent the same number of values.
- The difference is *what* those values are. Notice for example that `int` can represent $[-2^{31}, 0)$, while `unsigned` can represent $(2^{31} - 1, 2^{32} - 1]$, while *both* can represent $[0, 2^{31} - 1]$.
- This means that we have to think carefully when mixing these types.

# Data types

Fundamental data types

- Integer types
- **Character types**
- Floating-point types
- Other types

# Data types

Character types

- Very similar to integer types, but they are meant to represent characters in strings.
- Guranteed to be one byte (*at least* 8 bits).
- `signed char` - A character: can be negative.
- `unsigned char` - The smallest addressable unit.

# Data types

## Character types

- Very similar to integer types, but they are meant to represent characters in strings.
- Guranteed to be one byte (*at least* 8 bits).
- `signed char` - *Usually* in the range $[-128, 127]$.
- `unsigned char` - *Usually* in the range $[0, 255]$.

# Data types

Other character types

- `wchar_t` - Used for wide strings (32-bit chars)
- `char8_t` - Used for UTF-8 characters
- `char16_t` - Used for UTF-16 characters
- `char32_t` - Used for UTF-32 characters

# Data types

Fundamental data types

- Integer types
- Character types
- **Floating-point types**
- Other types

# Data types

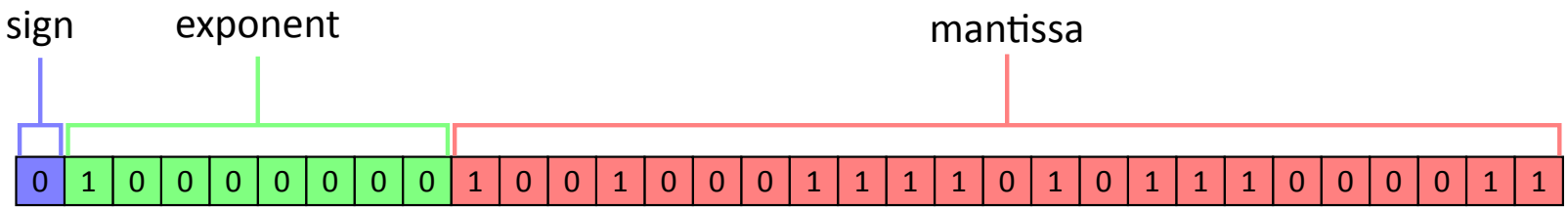
## Floating-point types

Floating-point numbers aren't supported on all systems. But if they are they must follow the specified standards:

- `float` - IEEE-754 binary32
- `double` - IEEE-754 binary64
- `long double` - Must support *one* of these standards:
  - IEEE-754 binary128
  - IEEE-754 binary64-extended

# Data types

## Floating-point types

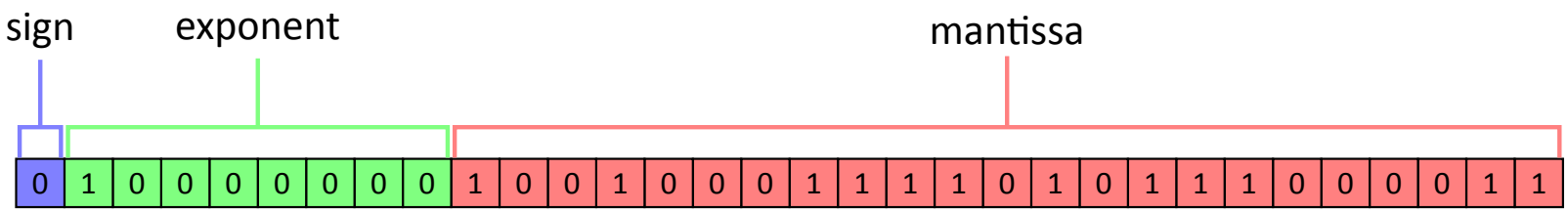



$$(-1)^{\texttt{sign}_2} \cdot 2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

This is the representation of floating-point numbers (`float`) inside the CPU. The general idea is that we have an integer where we can move around the decimal point inside to get any size of the number we want. There are three sections in the bits of a floating-point number: the sign bit, the exponent and the mantissa.

# Data types

## Floating-point types

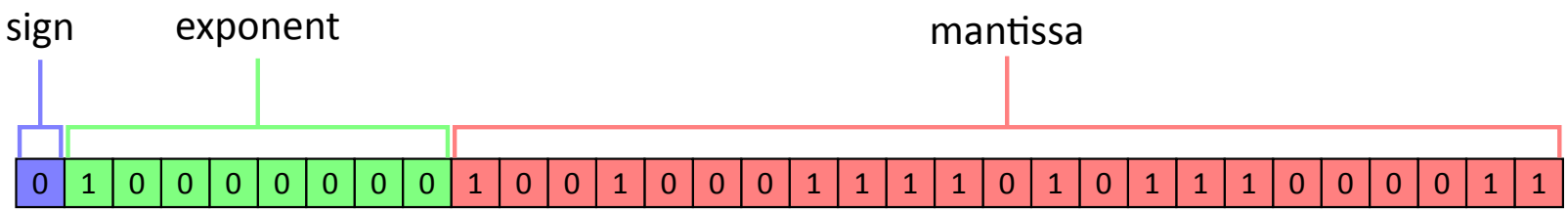



$$(-1)^{\texttt{sign}_2} \cdot 2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

The sign bit represents whether the number is positive or negative. This is achieved by raising -1 to that power, if `sign` = 1 then we get -1 and +1 otherwise.

# Data types

## Floating-point types

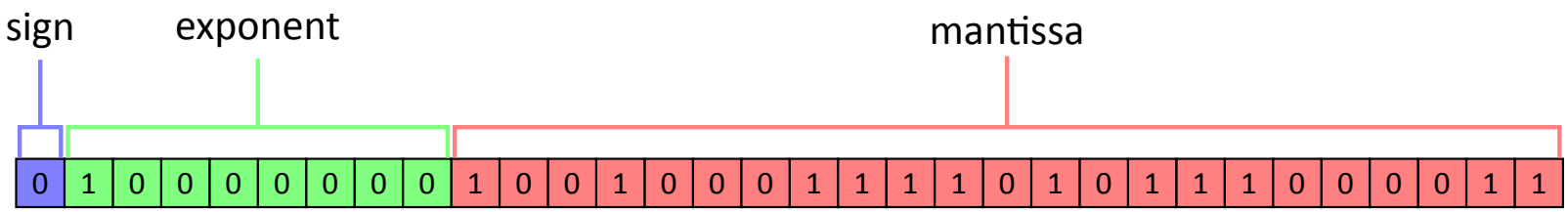



$$(-1)^{\texttt{sign}_2} \cdot 2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

We move the decimal point around inside the binary representation by multiplying by powers of $2$. Negative powers move the decimal point to the left, while positive powers move the decimal point to the right. This will therefore allow us to represent very large but also very small values.

# Data types

## Floating-point types

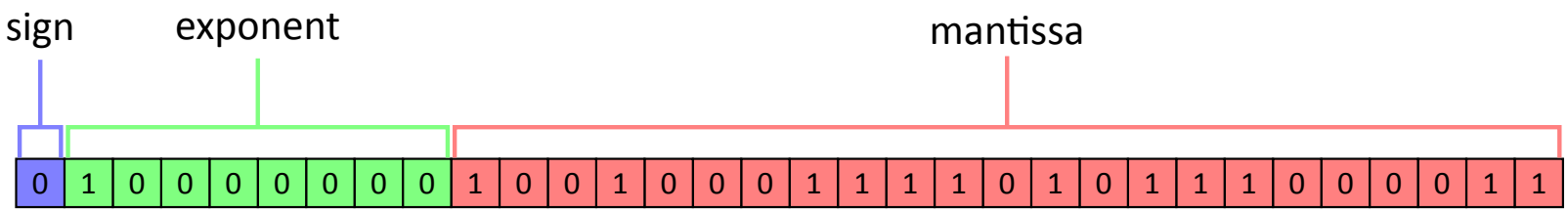



$$(-1)^{\texttt{sign}_2} \cdot 2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

To give access to both positive and negative powers, we let the exponent be an integer in $[0, 255]$ and then subtract $127$. This way we have a consistent representation of the exponent.

# Data types

## Floating-point types

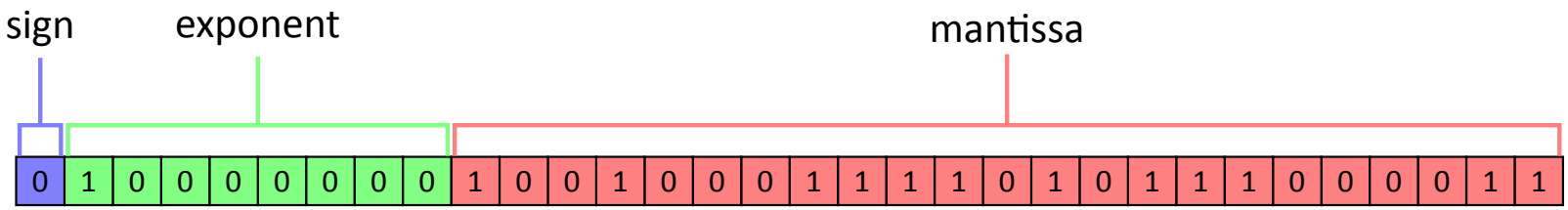



$$(-1)^{\texttt{sign}_2} \cdot 2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

The real number in the range $[1, 2)$ will always have the integer part $1$, so we do not explicitly store that, instead we just store the fractional part, which we call the mantissa. See the example on the next few slides.

# Data types

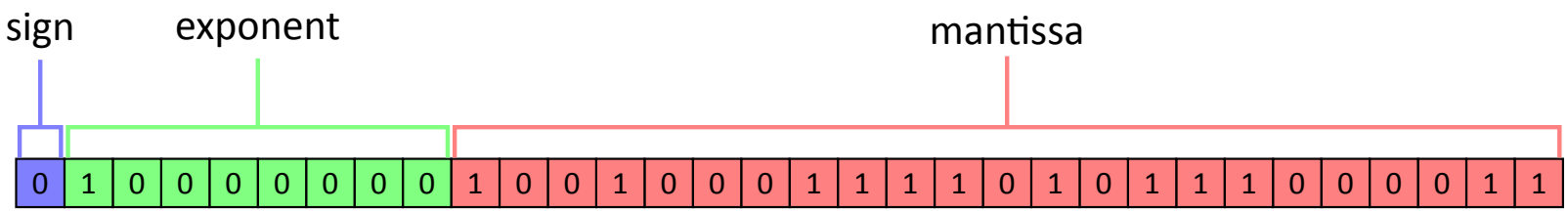
## Floating-point types




$$(-1)^{\texttt{sign}_2} \cdot 2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

# Data types

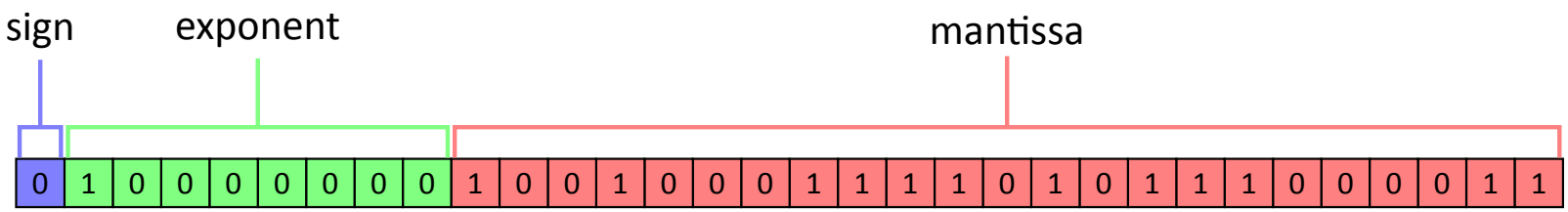
## Floating-point types




$$(-1)^{0_2} \cdot 2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

# Data types

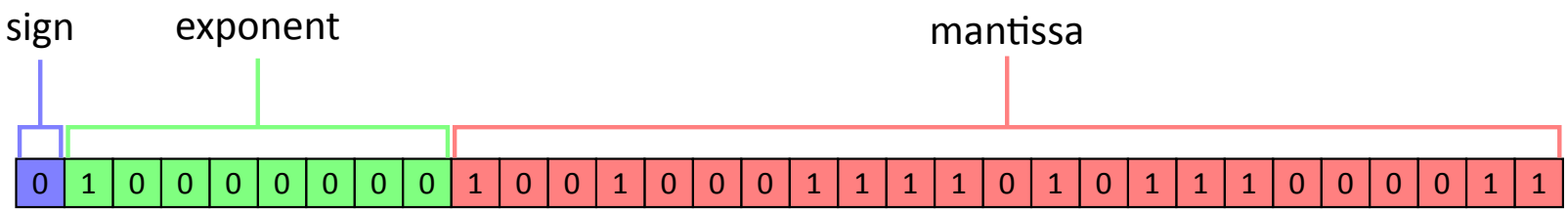
## Floating-point types




$$1 \cdot 2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

# Data types

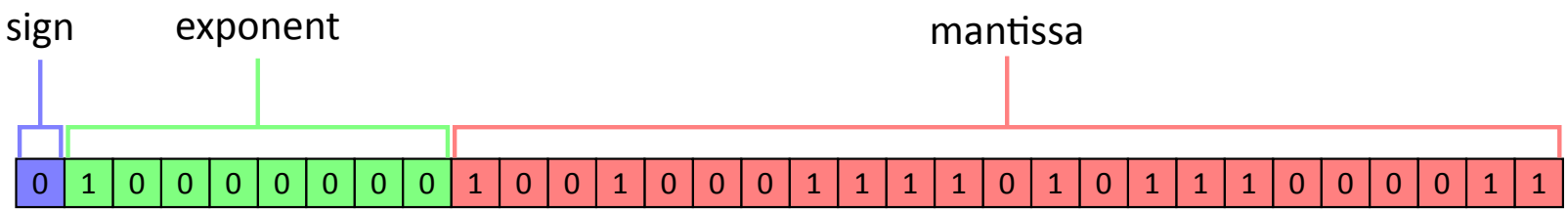
## Floating-point types




$$2^{\texttt{exponent}_2 - 127} \cdot (1.\texttt{mantissa}_2)$$

# Data types

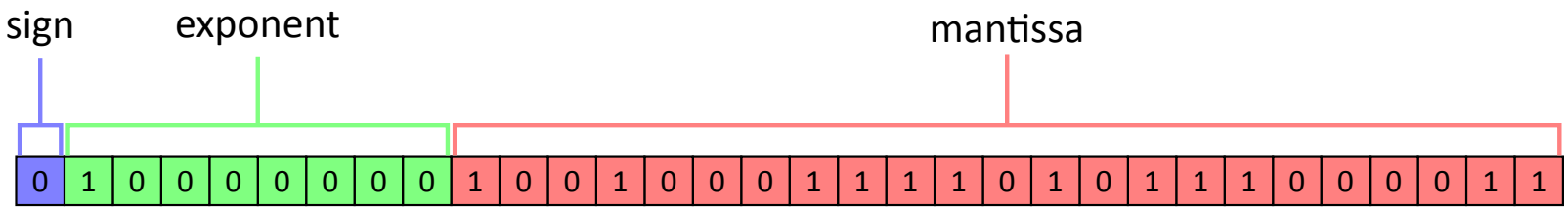
## Floating-point types




$$2^{10000000_2-127} \cdot (1.\texttt{mantissa}_2)$$

# Data types

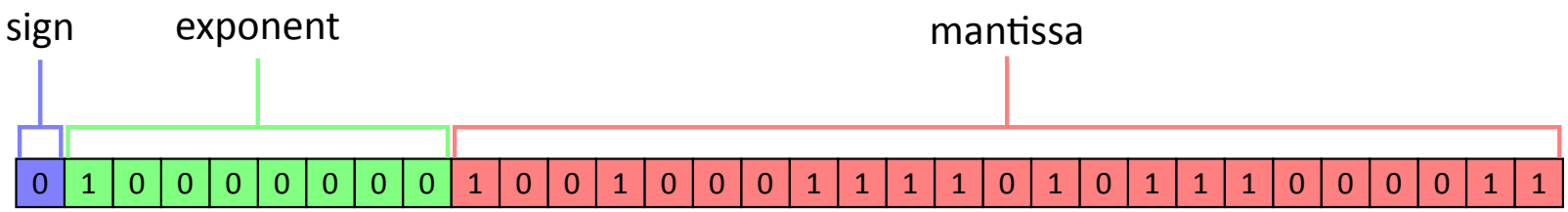
## Floating-point types




$$2^{128-127} \cdot (1.\texttt{mantissa}_2)$$

# Data types

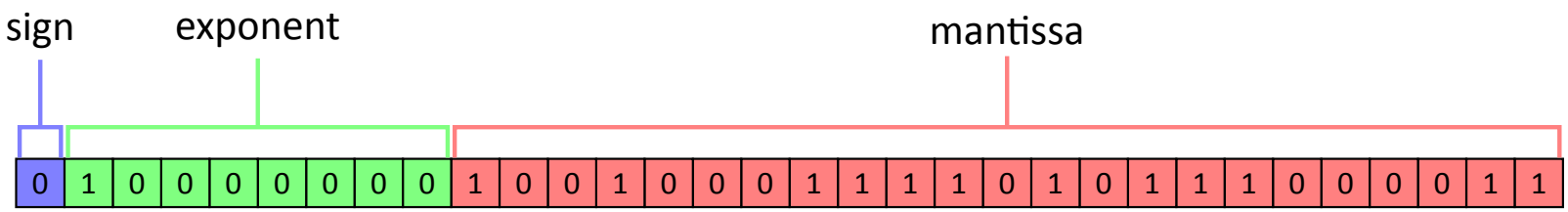
## Floating-point types




$$2^{1} \cdot (1.\texttt{mantissa}_2)$$

# Data types

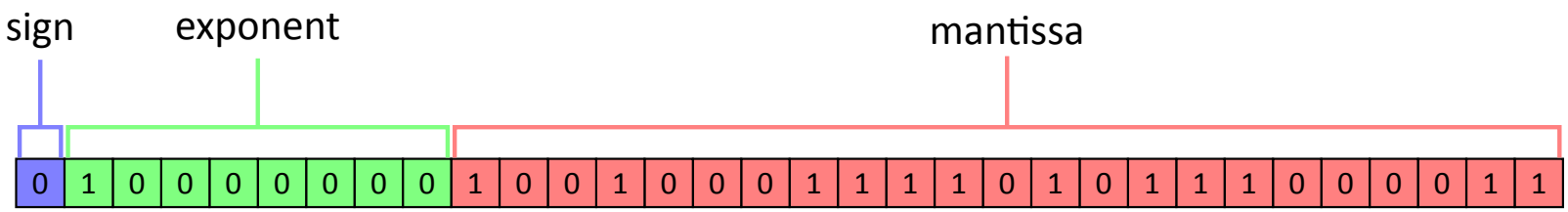
## Floating-point types




$$2 \cdot (1.\texttt{mantissa}_2)$$

# Data types

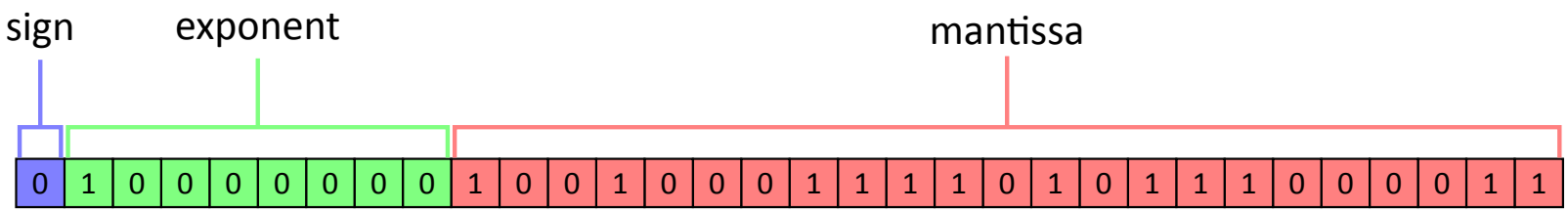
## Floating-point types




$$2 \cdot (1.10010001111010111000011_2)$$

# Data types

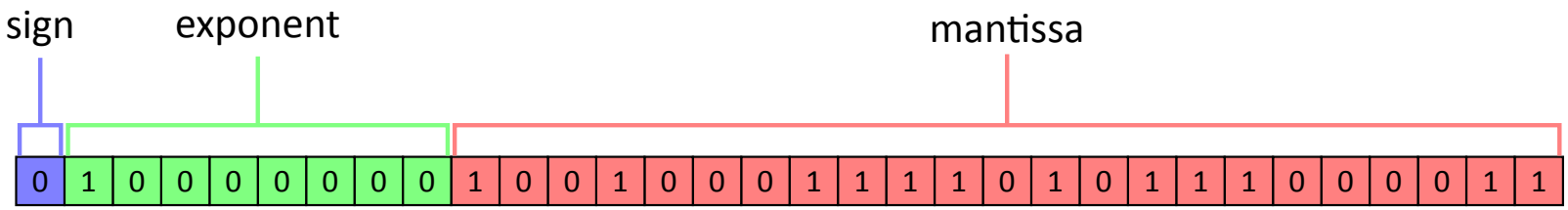
## Floating-point types




$$2 \cdot 1.5700000524520874$$

# Data types

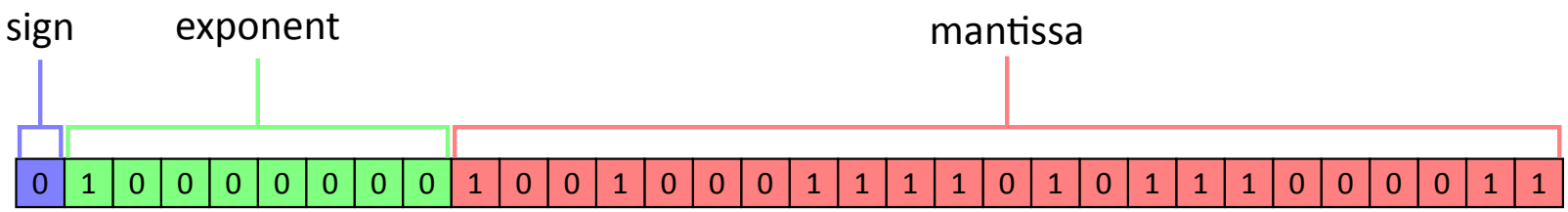
## Floating-point types




$$2 \cdot 1.5700000524520874$$

# Data types

## Floating-point types




$$\approx 3.14$$

# Data types

Floating-point types

- `float` has an 8-bit exponent and a 23-bit mantissa.
- `double` has an 11-bit exponent and a 52-bit mantissa.
- `long double` have different sizes on different systems and architectures.

# Data types

Fundamental data types

- Integer types
- Character types
- Floating-point types
- **Other types**

# Data types

Other fundamental types

- `bool` - `true` or `false`; is one byte in size
- `void` - a type without any values
- `std::nullptr_t` - the type of `nullptr`

# Data types

Data type categories

- Fundamental types
- **Array types**
- Enum types
- Class types (later)
- Pointer/Reference types (later)

# Data types

Array types

```
Type array[size]
```

# Data types

Array types

```
Type array[size]
```

The type of all the elements

# Data types

Array types

```
Type array[size]
```

This is the name of our array

# Data types

Array types

```
Type array[size]
```

The number of elements

# Data types

Array types

- These are called “C-Arrays”, so named because they where inherited from C.
- This allows us to create *fixed-sized* arrays, i.e. arrays where the size never change.
- The size (the number of elements) in the array **must** be known during compilation.

# Data types

Array types example

```
1  int array[3] { 1, 2, 3 };
2
3  array[0] = 2;
4  array[2] = array[2] - 1;
5
6  for (unsigned i { 0 }; i < 3; ++i)
7  {
8    std::cout << array[i] << std::endl;
9  }
```

# Data types

Array types example

This size is known during compilation, so its OK!

```
1  int array[3] { 1, 2, 3 };
2
3  array[0] = 2;
4  array[2] = array[2] - 1;
5
6  for (unsigned i { 0 }; i < 3; ++i)
7  {
8    std::cout << array[i] << std::endl;
9  }
```

# Data types

Array types example

```
1 int size { };
2 std::cout << "Enter size: ";
3 std::cin >> size;
4
5 int array[size] { };
```

# Data types

Array types example

```
1  int size { };
2  std::cout << "Enter size: ";
3  std::cin >> size;
4
5  int array[size] { };
```

You need to run this program to actually get the size...

# Data types

Array types example

```
1  int size { };
2  std::cout << "Enter size: ";
3  std::cin >> size;
4
5  int array[size] { };
```

... so the size is **not** known during compilation, which is **not** OK...

# Data types

Array types example

```
1  int size { };
2  std::cout << "Enter size: ";
3  std::cin >> size;
4
5  int array[size] { };
```

... or at least not if we follow standard C++. Some compilers *might* allow this.

# Data types

Array types example

```
1  int size { };
2  std::cout << "Enter size: ";
3  std::cin >> size;
4
5  int array[size] { };
```

Forbidden!

... or at least not if we follow standard C++. Some compilers *might* allow this.

# Data types

Array types example

- The compiler needs to know when *compiling* the program how much memory an array requires.
- This means that the number of elements must be known at *compile-time*.
- **However:** Some compilers have extensions that allow dynamically sized arrays, but standard C++ does not allow it.

# Data types

Data type categories

- Fundamental types
- Array types
- **Enum types**
- Class types (later)
- Pointer/Reference types (later)

# Data types

## Enumeration types

- An enumeration type (`enum`) is a type with a *discrete* set of named values.
- Each `enum` has an underlying *integer* representation, where each named value is assigned a specific value.
- Whenever an `enum` value is referenced in your code, the compiler translates that to the corresponding integer value.

# Data types

## Enumeration types

```
1  enum Direction
2  {
3    UNKNOWN, // = 0
4    NORTH,   // = 1
5    EAST,    // = 2
6    SOUTH,   // = 3
7    WEST     // = 4
8  };
```

# Data types

Enumeration types

```
1  enum Direction
2  {
3    UNKNOWN, // = 0
4    NORTH,   // = 1
5    EAST,    // = 2
6    SOUTH,   // = 3
7    WEST     // = 4
8  };
```

The code

```
1  Direction dir { NORTH };
2  switch (dir)
3  {
4    case NORTH: /* ... */ break;
5    case EAST:  /* ... */ break;
6    case SOUTH: /* ... */ break;
7    case WEST:  /* ... */ break;
8  }
```

This type of code is prefered!

# Data types

Enumeration types

```
1  enum Direction
2  {
3    UNKNOWN, // = 0
4    NORTH,   // = 1
5    EAST,    // = 2
6    SOUTH,   // = 3
7    WEST     // = 4
8  };
```

What the compiler sees

```
1  int dir { 1 };
2  switch (dir)
3  {
4    case 1: /* ... */ break;
5    case 2: /* ... */ break;
6    case 3: /* ... */ break;
7    case 4: /* ... */ break;
8  }
```

Don’t write code like this...

# Data types

Enumeration types

- The first value in an `enum` is assigned the value `0`.
- Each other value is assigned the previous value + 1.
- It is possible to control the underlying values yourself.
- This is useful in situations where you need the `enum` values to correspond with specific integer values (for example when working with *bit flags*).
- You can also specify what type should be used for representing the `enum`.

# Data types

Enumeration types

```
1  enum Log_Level : char
2  {
3
4    DEBUG   = 0b0001, // 1
5    INFO    = 0b0010, // 2
6    WARNING = 0b0100, // 4
7    ERROR   = 0b1000  // 8
8
9  };
```

# Data types

Enumeration types

```
1  enum Log_Level : char
2  {
3
4    DEBUG   = 0b0001, // 1
5    INFO    = 0b0010, // 2
6    WARNING = 0b0100, // 4
7    ERROR   = 0b1000  // 8
8
9  };
```

The code

```
1  Log_Level active {
2    INFO | WARNING | ERROR
3  };
4
5  if (active & DEBUG)
6    // write DEBUG logs
7  else if (active & INFO)
8    // write INFO logs
9  // ...
```

This type of code is prefered!

# Data types

Enumeration types

```
1  enum Log_Level : char
2  {
3
4    DEBUG   = 0b0001, // 1
5    INFO    = 0b0010, // 2
6    WARNING = 0b0100, // 4
7    ERROR   = 0b1000  // 8
8
9  };
```

What the compiler sees

```
1  char active {
2   0b0010 | 0b0100 | 0b1000
3  };
4
5  if (active & 0b0001)
6    // write DEBUG logs
7  else if (active & 0b0010)
8    // write INFO logs
9  // ...
```

Don’t write code like this...

# Data types

Enumeration types

- `Log_Level` is represented as a `char`.
- In this example we use `Log_Level` as a *bitfield*, meaning that each value in the `enum` represent a unique bit.
- Every literal that starts with `0b` is a *binary* number.
- we *combine* values using *bitwise or* (`|`).
- We check if a bit is set using *bitwise and* (`&`).

# Data types

A problem with enumeration types

```
1  enum Log_Level : char
2  {
3
4    DEBUG   = 0b0001, // 1
5    INFO    = 0b0010, // 2
6    WARNING = 0b0100, // 4
7    ERROR   = 0b1000  // 8
8
9  };
```

```
1  enum Status
2  {
3
4    PENDING,   //  0
5    ACCEPTED,  //  1
6    DENIED,    //  2
7    ERROR = -1 // -1
8
9  };
```

# Data types

A problem with enumeration types

```
1  enum Log_Level : char
2  {
3
4    DEBUG   = 0b0001, // 1
5    INFO    = 0b0010, // 2
6    WARNING = 0b0100, // 4
7    ERROR   = 0b1000  // 8
8
9  };
```

```
1  enum Status
2  {
3
4    PENDING,   //  0
5    ACCEPTED,  //  1
6    DENIED,    //  2
7    ERROR = -1 // -1
8
9  };
```

# Data types

A problem with enumeration types

```
1  enum Log_Level : char
2  {
3
4    DEBUG   = 0b0001, // 1
5    INFO    = 0b0010, // 2
6    WARNING = 0b0100, // 4
7    ERROR   = 0b1000  // 8
8
9  };
```

```
1  enum Status
2  {
3
4    PENDING,    //  0
5    ACCEPTED,   //  1
6    DENIED,     //  2
7    ERROR = -1 // -1
8
9  };
```

Ambiguous!

# Data types

A problem with enumeration types

- If we have two enums which have values that share a name, then whenever we reference that name it will be ambiguous.
- In C the problem was solved by adding a prefix to all values. For example: `LOG_LEVEL_ERROR`, but this is generally a pretty terse solution.
- In C++ we can use something called *scoped enum*.

# Data types

Scoped enums

```
1  enum class Status
2  {
3
4    PENDING,
5    ACCEPTED,
6    DENIED,
7    ERROR = -1
8
9  };
```

# Data types

## Scoped enums

```
1  enum class Status
2  {
3
4    PENDING,
5    ACCEPTED,
6    DENIED,
7    ERROR = -1
8
9  };
```

Use keywords `class` or `struct` to declare a *scoped enum*.

# Data types

Scoped enums

```
1  enum struct Status
2  {
3
4    PENDING,
5    ACCEPTED,
6    DENIED,
7    ERROR = -1
8
9  };
```

Use keywords `class` or `struct` to declare a *scoped enum*.

# Data types

Scoped enums

```
1  enum struct Status
2  {
3
4    PENDING,
5    ACCEPTED,
6    DENIED,
7    ERROR = -1
8
9  };
```

# Data types

Scoped enums

```
1  enum struct Status
2  {
3
4    PENDING,
5    ACCEPTED,
6    DENIED,
7    ERROR = -1
8
9  };
```

```
1  // PENDING is default
2  Status status { };
3  while (status == Status::PENDING)
4  {
5    status = handle();
6    if (status == Status::DENIED)
7      // ...
8    // ...
9  }
```

# Data types

Scoped enums

```
1  enum struct Status
2  {
3
4    PENDING,
5    ACCEPTED,
6    DENIED,
7    ERROR = -1
8
9  };
```

```
1  // PENDING is default
2  Status status { };
3  while (status == Status::PENDING)
4  {
5    status = handle();
6    if (status == Status::DENIED)
7      // ...
8    // ...
9  }
```

Whenever we reference a value inside a *scoped enum* we use `::`

# Data types

Scoped enums

```
1  enum struct Status
2  {
3
4    PENDING,
5    ACCEPTED,
6    DENIED,
7    ERROR = -1
8
9  };
```

```
1
2  // Not ambiguous since Status
3  // is a scoped enum!
4
5  Status status { Status::ERROR };
6
7  Log_Level level { ERROR };
8
9
```

# Data types

CV-qualifiers

```
1 int var { 5 };
2 var = 7;
3 std::cout << var << std::endl;
```

# Data types

CV-qualifiers

```
1 int var { 5 };
2 var = 7;
3 std::cout << var << std::endl;
```

Ok!

# Data types

CV-qualifiers

```
1 int const var { 5 };
2 var = 7;
3 std::cout << var << std::endl;
```

# Data types

CV-qualifiers

```
1 int const var { 5 };
2 var = 7;
3 std::cout << var << std::endl;
```

# Data types

CV-qualifiers

```
1 int const var { 5 };
2 var = 7;
3 std::cout << var << std::endl;
```

Compile Error!

# Data types

CV-qualifiers

- CV-qualifiers are modifiers that apply to types. There are two CV-qualifiers:
- `const` which makes the object *constant* (meaning you cannot modify it after creation)
- `volatile` which marks the object as *volatile*. This means that any read or write of the variable counts as a *visible side-effect*.
- **Note:** `volatile` will not be covered in this course, but `const` will!

# Data types

CV-qualifiers

- CV-qualifiers counts as being a part of the type. So `int const` is a different type compared to `int`.
- There are two ways to apply CV-qualifiers...

# Data types

CV-qualifiers

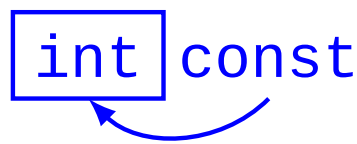
**Rule of thumb:** CV-qualifiers applies to the left:

```
int const
```

# Data types

CV-qualifiers

**Rule of thumb:** CV-qualifiers applies to the left:



# Data types

CV-qualifiers

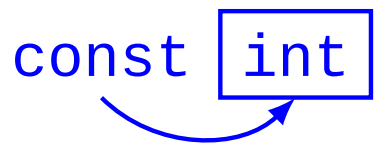
... except when there is nothing to the left:

```
const int
```

# Data types

CV-qualifiers

... except when there is nothing to the left:



# Data types

CV-qualifiers

This is relevant for more complicated type declarations:

```
int const * const
```

# Data types

CV-qualifiers

This is relevant for more complicated type declarations:


A *constant* pointer...

# Data types

CV-qualifiers

This is relevant for more complicated type declarations:


... to a *constant* `int`!

# Data types

CV-qualifiers

In comparison to:

```
const  int * const
```

# Data types

CV-qualifiers

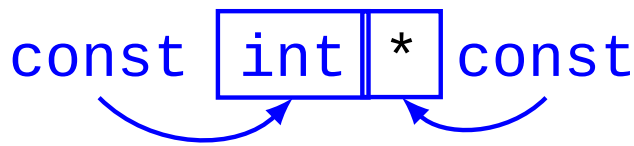
In comparison to:


A *constant* pointer...

# Data types

CV-qualifiers

In comparison to:




... to a *constant* `int`?

# Data types

CV-qualifiers

**Conclusion:** Always put `const` to the *right* of whatever you want it to apply to, this way it is easier to understand!

# Data types

C-strings

```
1 /* what type? */ str { "Hello" };
```

# Data types

C-strings

```
1 char str[6] { "Hello" };
```

# Data types

C-strings

```
1 char str[6] { 'H', 'e', 'l', 'l', 'o', '\0' };
```

# Data types

C-strings

```
1 char str[6] { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Null-terminator

# Data types

C-strings

- To use `std::string` we have to include `<string>`
- But we can still use *string literals*, for example: `"Hello"` without including `<string>`. How come?
- String literals aren't `std::string`. They are so-called *C-strings* (since this is how strings work i C)
- A *C-string* is just a `char` array which ends with the special character `'\0'` (called the *null-terminator*).
- The null-terminator is used to mark the end of a string.

1 Data types
2 Functions
3 Conversions
4 Initialization

# Functions

Function definition

What type this function returns (the *return type*)

```
1 int add(int a, int b)
2 {
3   return a + b;
4 }
```

# Functions

Function definition

The *name* of the function

```
1  int add(int a, int b)
2  {
3    return a + b;
4  }
```

# Functions

Function definition

All the input *parameters* to the function

```
1  int add(int a, int b)
2  {
3    return a + b;
4  }
```

# Functions

Function definition

The implementation of the function (the *function body*)

```
1  int add(int a, int b)
2  {
3    return a + b;
4  }
```

# Functions

Multiple overloads!

```
1  int add(int a, int b)
2  {
3    return a + b;
4  }
5
6  double add(double a, double b)
7  {
8    return a + b;
9  }
10
11 int add(int a, int b, int c)
12 {
13   return a + b + c;
14 }
```

# Functions

Single pass

- C++ is what is sometimes refered to as *single-pass*
- This means that compilers only have to go through the code top-to-bottom *once*
- A consequence of this is that at any given point in the source code, the compiler only knows about the things that are declared at that point.
- On the next slide we'll see a problem caused by this...

# Functions

Declaration & Definition

```
 1 int add(int a, int b)
 2 {
 3   if (b < 0)
 4     return sub(a, -b);
 5   return a + b;
 6 }
 7
 8 int sub(int a, int b)
 9 {
10   if (b < 0)
11     return add(a, -b);
12   return a - b;
13 }
```

# Functions

Declaration & Definition

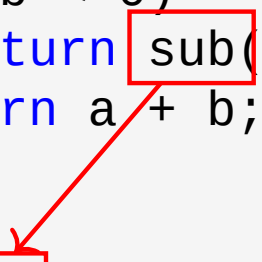
```
1  int add(int a, int b)
2  {
3    if (b < 0)
4      return sub(a, -b);
5    return a + b;
6  }
7
8  int sub(int a, int b)
9  {
10   if (b < 0)
11     return add(a, -b);
12   return a - b;
13 }
```

Here the compiler finds a call to `add()` *after* it has seen the definition, which is OK!

# Functions

Declaration & Definition

```
1  int add(int a, int b)
2  {
3    if (b < 0)
4      return sub(a, -b);
5    return a + b;
6  }
7
8  int sub(int a, int b)
9  {
10   if (b < 0)
11     return add(a, -b);
12   return a - b;
13 }
```



But the compiler finds a call to sub( ) *before* it has been defined, which is **not** OK...

# Functions

Declaration & Definition

```
1  int add(int a, int b)
2  {
3    if (b < 0)
4      return sub(a, -b);
5    return a + b;
6  }
7
8  int sub(int a, int b)
9  {
10   if (b < 0)
11     return add(a, -b);
12   return a - b;
13 }
```

Compile Error!

# Functions

Declaration & Definition

```
 1 int sub(int a, int b)
 2 {
 3   if (b < 0)
 4     return add(a, -b);
 5   return a - b;
 6 }
 7
 8 int add(int a, int b)
 9 {
10   if (b < 0)
11     return sub(a, -b);
12   return a + b;
13 }
```

# Functions

Declaration & Definition

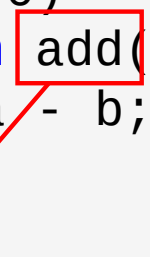
```
1  int sub(int a, int b)
2  {
3    if (b < 0)
4      return add(a, -b);
5    return a - b;
6  }
7
8  int add(int a, int b)
9  {
10   if (b < 0)
11     return sub(a, -b);
12   return a + b;
13 }
```

Now `sub( )` is defined *before* it is called!

# Functions

Declaration & Definition

```
1  int sub(int a, int b)
2  {
3    if (b < 0)
4      return add(a, -b);
5    return a - b;
6  }
7
8  int add(int a, int b)
9  {
10   if (b < 0)
11     return sub(a, -b);
12   return a + b;
13 }
```



But now add( ) is the problem instead...

# Functions

Declaration & Definition

```
1  int sub(int a, int b)
2  {
3    if (b < 0)
4      return add(a, -b);
5    return a - b;
6  }
7
8  int add(int a, int b)
9  {
10   if (b < 0)
11     return sub(a, -b);
12   return a + b;
13 }
```

Compile Error!

# Functions

## Declaration & Definition

```
1  int sub(int a, int b);
2  int add(int a, int b);
3
4  int add(int a, int b)
5  {
6    if (b < 0)
7      return sub(a, -b);
8    return a + b;
9  }
10
11 int sub(int a, int b)
12 {
13   if (b < 0)
14     return add(a, -b);
15   return a - b;
16 }
```

We declare the functions before defining them! That way the compiler will know about the functions when defining other functions.

# Functions

## Overload resolution

```
1  // Suppose we have:
2  int print(int x) { /* ... */ }
3
4  int main()
5  {
6    print(3);    // works
7  }
```

# Functions

## Overload resolution

```
1  // Suppose we have:
2  int print(int x) { /* ... */ }
3
4  int main()
5  {
6    print(3.0);  // works?
7  }
```

# Functions

Overload resolution

```
1  // Suppose we have:
2  int print(int x) { /* ... */ }
3
4  int main()
5  {
6    print(true); // works?!
7  }
```

1 Data types
2 Functions
3 Conversions
4 Initialization

# Conversions

Implict conversion

- Arguments
- Operands
- Initializations
- Conditions

# Conversions

Implicit conversions

- In certain circumstances the compiler is allowed to silently convert values to other data types. This is refered to as *implicit conversion*.
- Implicit conversion can be done to values that occur as:
  - arguments to function calls
  - operands in expressions
  - initial values to variables
  - conditions in loops and if-statements

# Conversions

Implicit conversions

- Promotions
- Numeric conversions
- Boolean conversions
- Function-to-pointer conversion
- Array-to-pointer conversion
- Qualification conversion
- *lvalue-to-rvalue conversion* (later)

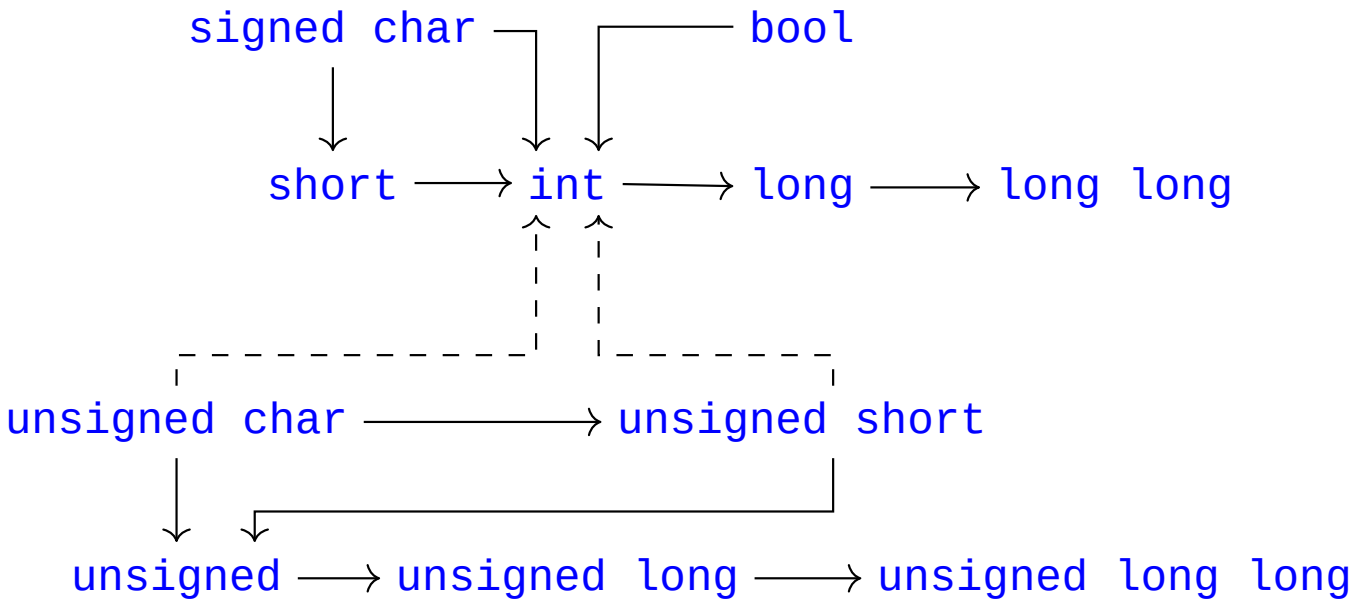
# Conversions

Implicit conversions

- **Promotions**
- Numeric conversions
- Boolean conversions
- Function-to-pointer conversion
- Array-to-pointer conversion
- Qualification conversion
- *lvalue-to-rvalue conversion* (later)

# Conversions

Integer promotions




A ⟶ B: A can be converted to B

A ⇢ B: This conversion is available on some architectures

# Conversions

Integer promotions

- An integer promotion is a type of *implicit conversion* that the compiler can perform on values of integer types.
- Promotions tend to happen in arithmetic expressions.
- `int` and `unsigned` are the smallest types which can appear in arithmeric operations.
- This means that `char + char` results in an `int` since both `char` values are converted to `int`.

# Conversions

## Integer promotions

- The diagram summarizes the integer promotion rules.
- The solid arrows means that the conversion can always happen.
- Dashed arrows means that this conversion can happen on *some systems*.
- The rule of thumb is that we can always promote to *larger* types.
- Specifically: if all the values of type A can be represented in type B, then A can be promoted to B.

# Conversions

## Integer promotions

- Note that the set of positive values is larger for `unsigned`. This means that there are some values which cannot be represented as the `signed` counter-part (and vice versa).
- **For example:** The value 192 can be represented in an 8-bit `unsigned char` but cannot be represented in an 8-bit `char`.
- This means that some conversions validity depends on the representation.
- **For example:** there might exist a system where both `unsigned short` and `int` are 16-bits, which means there are certain values in `unsigned short` that cannot be represented in `int` (for example 56341). But if the `int` is a 32-bit integer then all values of `unsigned short` *can* be represented as `int`.
- These cases are marked with dashed lines in the diagram.

# Conversions

Floating-point promotions

`float` $\longrightarrow$ `double` $\longrightarrow$ `long double`

# Conversions

Floating-point promotions

- Floating-point promotions are much more straight-forward compared to integer promotions.
- You just have to remember that `float` is smaller than `double`, which is smaller than `long double`.

# Conversions

Narrowing conversions

We refer to conversions in the opposite direction of the promotions as *narrowing conversions* since these are guaranteed to lose precision.

# Conversions

Implicit conversions

- Promotions
- Numeric conversions
- Boolean conversions
- Function-to-pointer conversion
- Array-to-pointer conversion
- Qualification conversion
- *lvalue-to-rvalue conversion* (later)

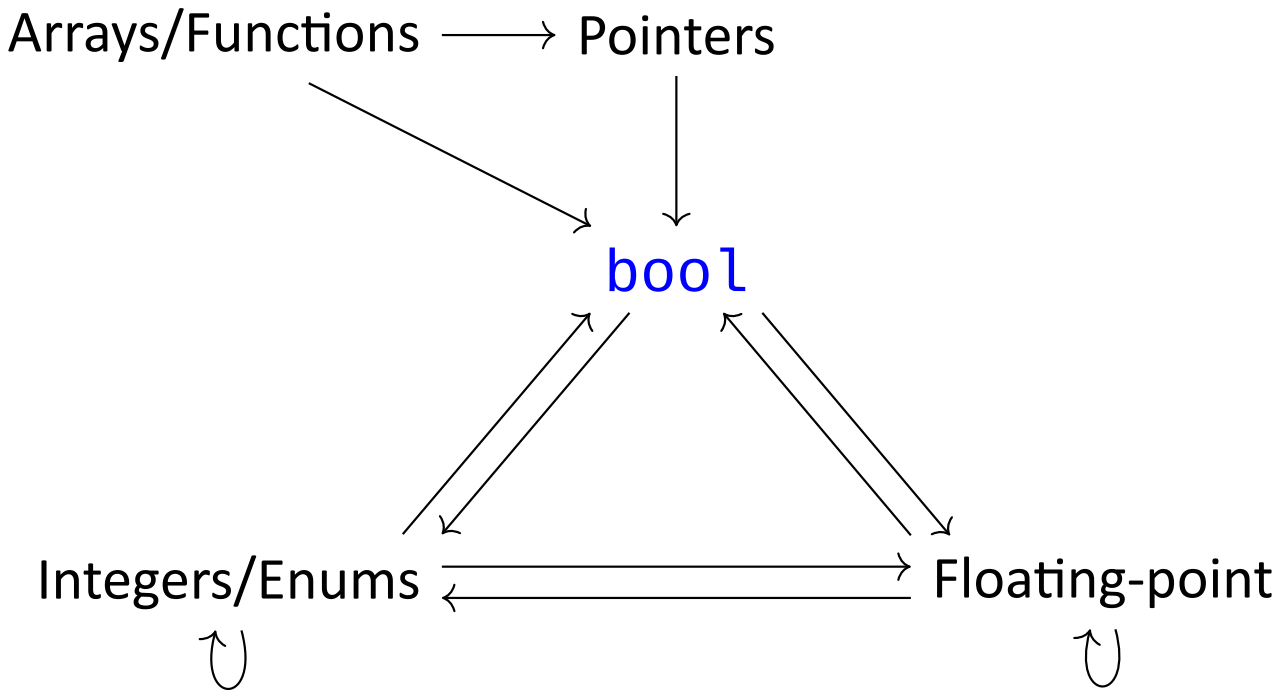
# Conversions

Implicit conversions

- Promotions
- **Numeric conversions**
- **Boolean conversions**
- **Function-to-pointer conversion**
- **Array-to-pointer conversion**
- Qualification conversion
- *lvalue-to-rvalue conversion* (later)

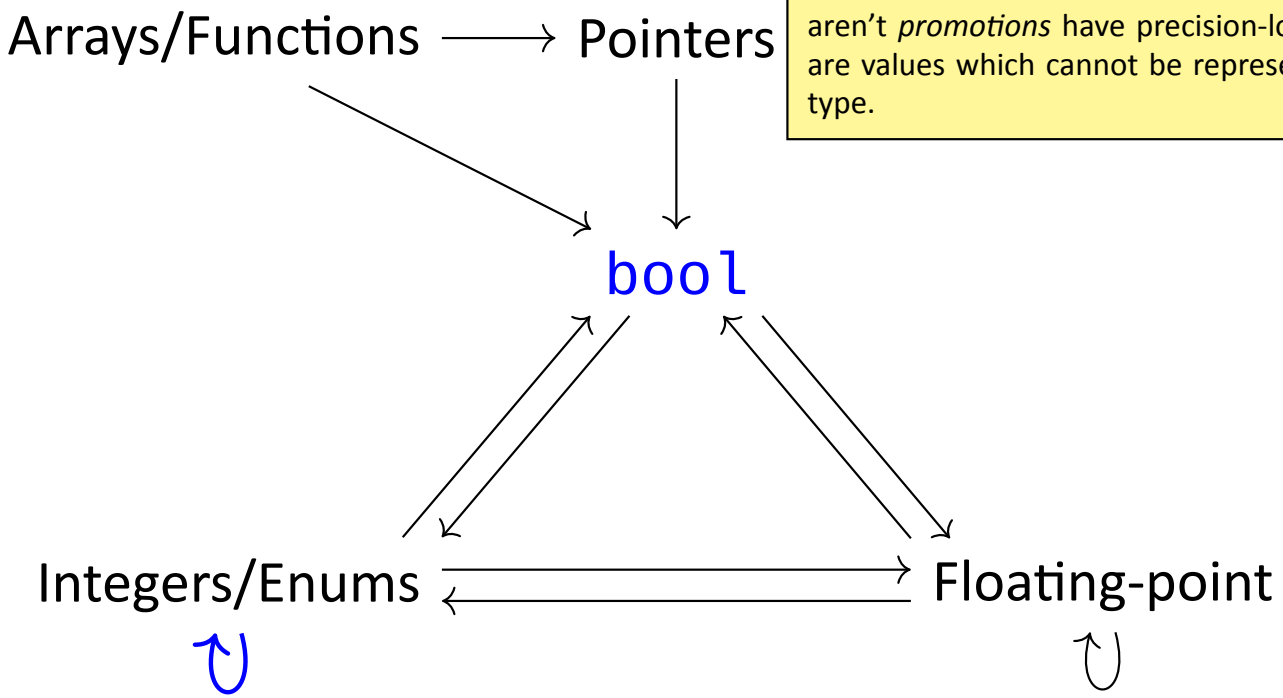
# Conversions

Numeric & Boolean conversions



# Conversions

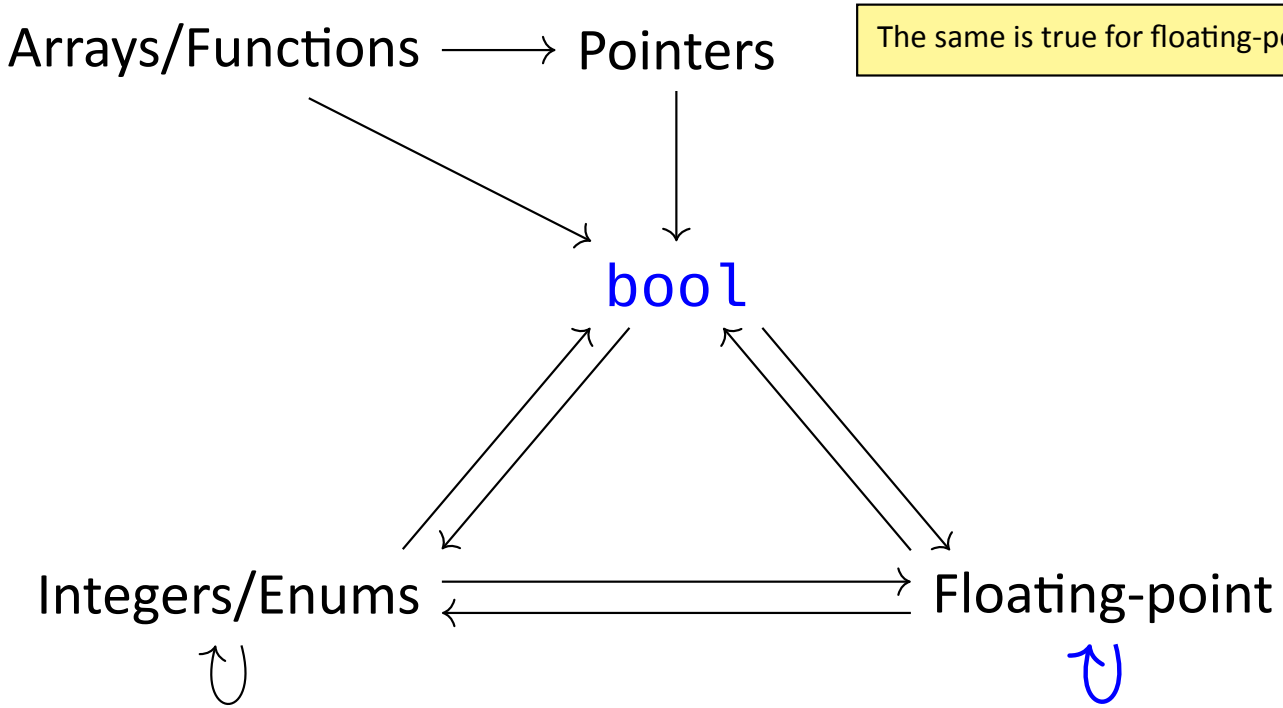
Numeric & Boolean conversions




Integers and enum types can all be *converted* between each other. However note that every conversion that aren't *promotions* have precision-loss, meaning there are values which cannot be represented in the target type.

# Conversions

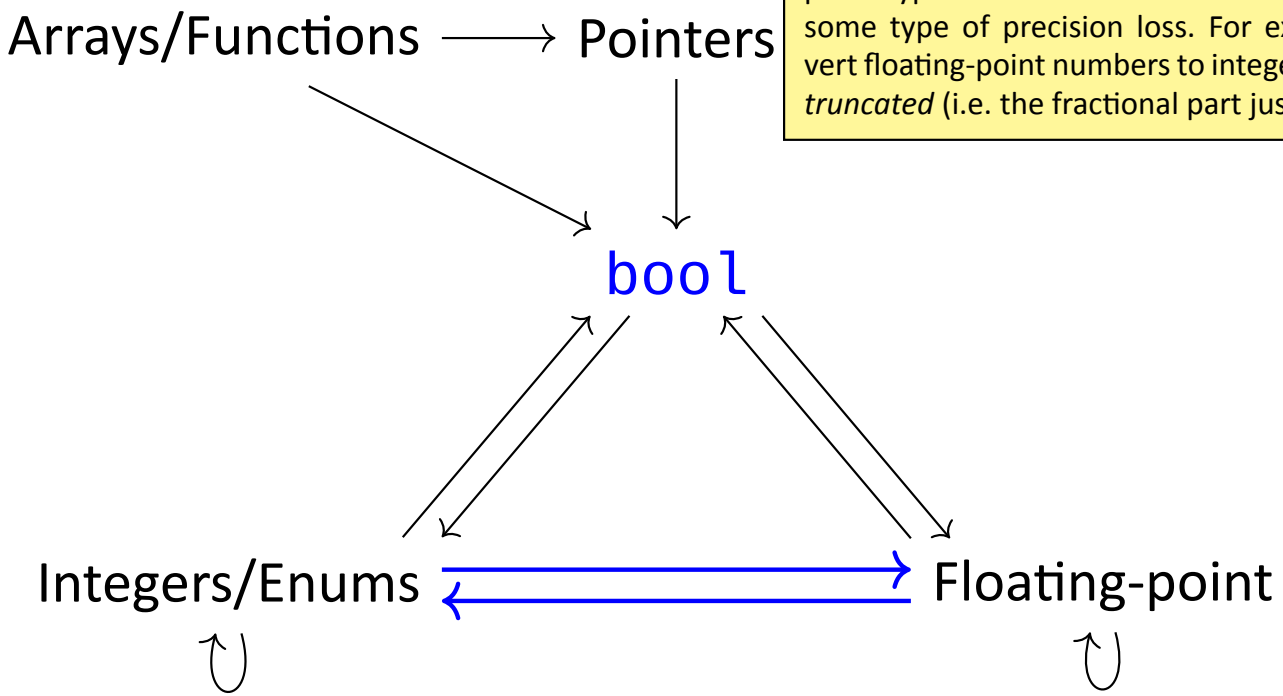
Numeric & Boolean conversions




The same is true for floating-point numbers.

# Conversions

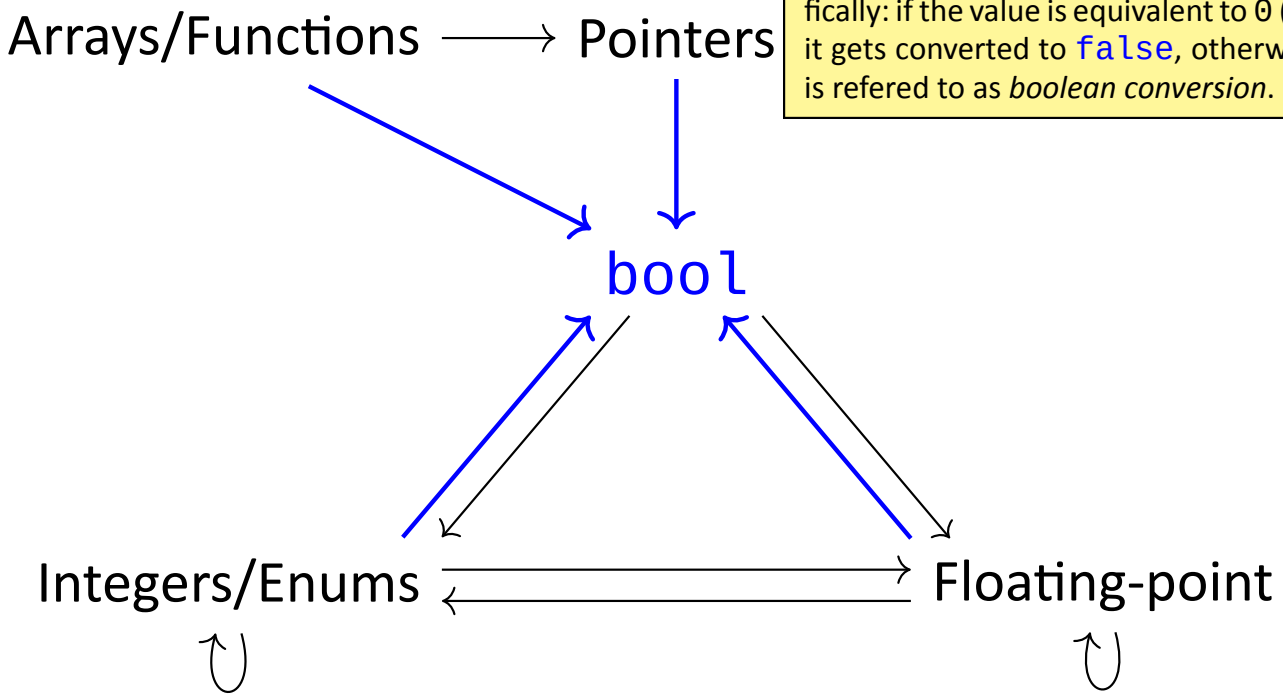
Numeric & Boolean conversions




All integer/enum types can be converted to *all* floating-point types and vice versa. However, this will lead to some type of precision loss. For example, if we convert floating-point numbers to integers the value will be *truncated* (i.e. the fractional part just gets removed).

# Conversions

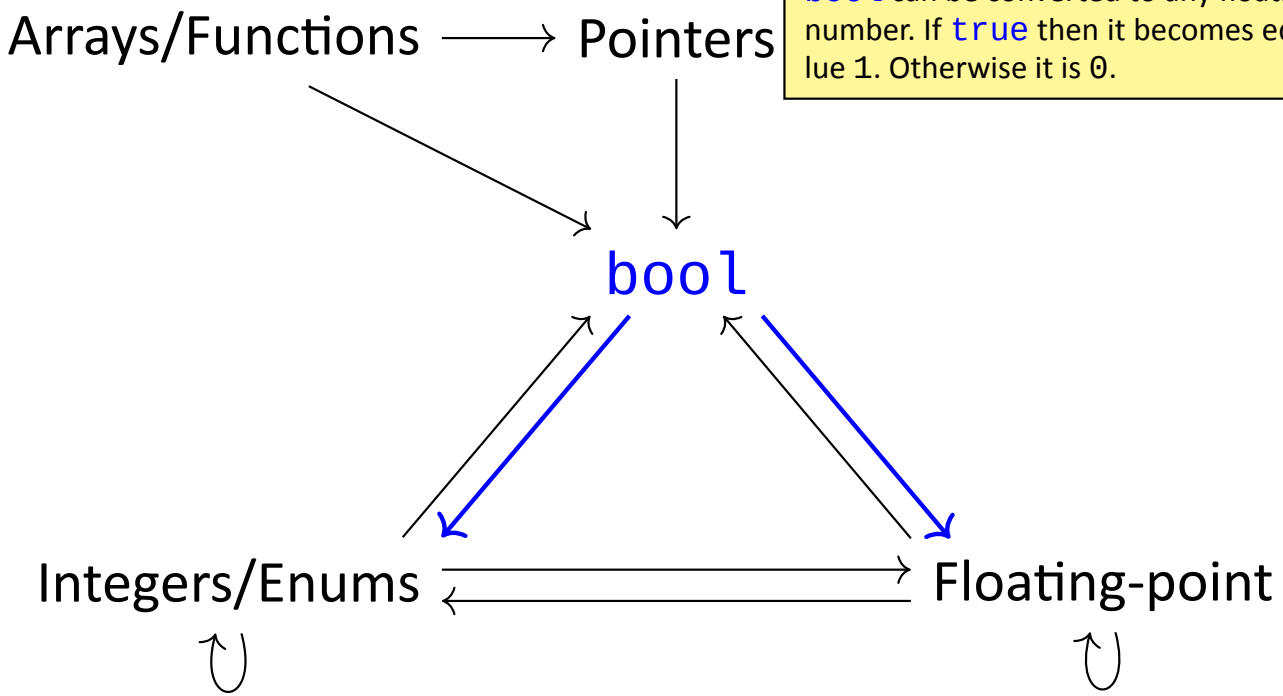
Numeric & Boolean conversions




All fundamental types can be converted to `bool`. Specifically: if the value is equivalent to `0` (or `nullptr`) then it gets converted to `false`, otherwise it is `true`. This is refered to as *boolean conversion*.

# Conversions

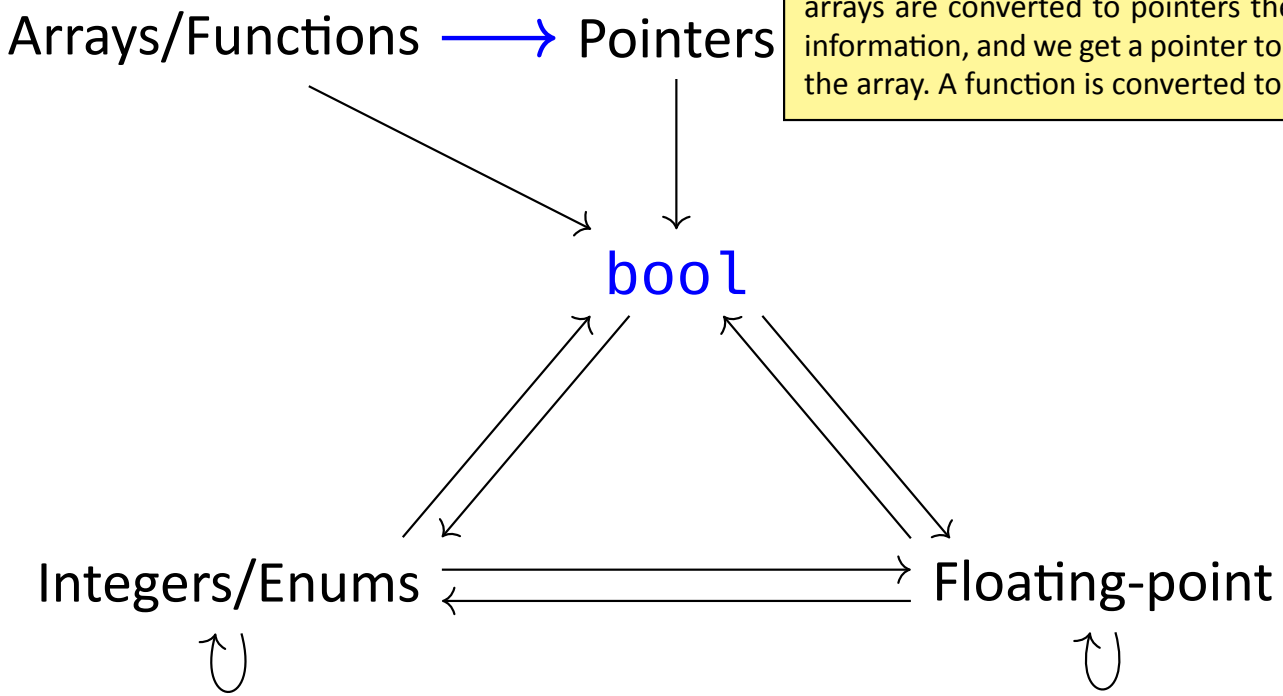
Numeric & Boolean conversions




`bool` can be converted to any floating-point or integer number. If `true` then it becomes equivalent to the value `1`. Otherwise it is `0`.

# Conversions

Numeric & Boolean conversions




Both arrays functions can be converted to pointers. If arrays are converted to pointers then we lose the size information, and we get a pointer to the first element in the array. A function is converted to a *function-pointer*.

# Conversions

Implicit conversions

- Promotions
- Numeric conversions
- Boolean conversions
- Function-to-pointer conversion
- Array-to-pointer conversion
- Qualification conversion
- *lvalue-to-rvalue conversion* (later)

# Conversions

Implicit conversions

- Promotions
- Numeric conversions
- Boolean conversions
- Function-to-pointer conversion
- Array-to-pointer conversion
- **Qualification conversion**
- *lvalue-to-rvalue conversion* (later)

# Conversions

Qualification conversion

- The compiler is allowed to *add* `const` (and `volatile`) as needed during a conversion.
- There are certain rules for when and how it is allowed to do this, you can read more on cppreference

# Conversions

Standard conversion sequence

Perform these conversions in-order (steps can be skipped):

1. *array-to-pointer*, *function-to-pointer*, or *lvalue-to-rvalue*
2. *numeric promotion* if possible, otherwise *numeric conversion*
3. *qualification conversion*

# Conversions

Explicit casts

Read the specified reading material here.

# Conversions

What will be printed?

```
1  int main()
2  {
3    int array[5] {1,2,3,4,5};
4    cout << array << endl;
5  }
```

# Conversions

What will be printed?

```
1  int main()
2  {
3    char str[4] {'h', 'i', '!', '\0'};
4    cout << str << endl;
5  }
```

# Conversions

What will be printed?

```
1  void foo() { cout << "foo" << endl; }
2
3  int main()
4  {
5    cout << foo << endl;
6  }
```

1 Data types
2 Functions
3 Conversions
4 Initialization

LINKÖPING UNIVERSITY

# Initialization

Ways of initialization

- Copy initialization: `int x = 5;`
- Value initialization: `int x{};`
- Direct initialization: `int x(5);`
- List initialization: `int x{5};`

# Initialization

Ways of initialization

- Copy initialization: `int x = 5;`
  - initialize an object by copying another object
  - will try to implicitly convert a value to make it work
  - tries to call any non-explicit constructors with one parameter
- Value initialization: `int x{};`
- Direct initialization: `int x(5);`
- List initialization: `int x{5};`

# Initialization

Ways of initialization

- Copy initialization: `int x = 5;`
- Value initialization: `int x{};`
  - call the *default constructor*
  - if no default constructor exists, it will default initialize the object (set all bytes to zero)
- Direct initialization: `int x(5);`
- List initialization: `int x{5};`

# Initialization

Direct vs. List initialization

What will they try to do?

*Direct initialization* ( )

1. appropriate constructor
2. aggregate initialization
3. copy initialization

Narrowing conversions are **allowed**.

*List initialization* {}

1. aggregate initialization
2. appropriate constructor
3. copy initialization

Narrowing conversions are **prohibited**.

# Initialization

Direct vs. List initialization

What will they try to do?

*Direct initialization* ( )

1. appropriate constructor
2. aggregate initialization
3. copy initialization

Narrowing conversions are **allowed**.

*List initialization* {}

1. aggregate initialization
2. appropriate constructor
3. copy initialization

Narrowing conversions are **prohibited**.

# Initialization

Direct vs. List initialization

What will they try to do?

*Direct initialization* ( )

1. appropriate constructor
2. aggregate initialization
3. copy initialization

Narrowing conversions are **allowed**.

*List initialization* {}

1. aggregate initialization
2. appropriate constructor
3. copy initialization

Narrowing conversions are **prohibited**.

List initialization is recommended

# Initialization

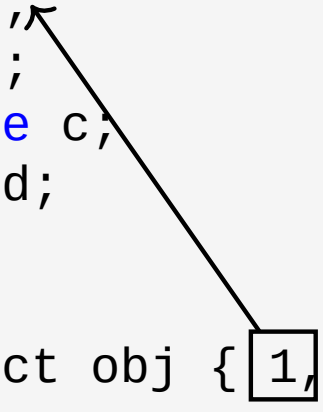
Aggregate initialization

```
1  struct My_Struct
2  {
3    int a;
4    int b;
5    double c;
6    char d;
7  };
8
9  My_Struct obj { 1, 2, 3.4, '5' };
```

# Initialization

Aggregate initialization

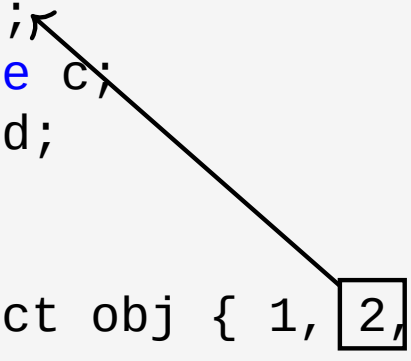
```
1  struct My_Struct
2  {
3    int a;
4    int b;
5    double c;
6    char d;
7  };
8
9  My_Struct obj { 1, 2, 3.4, '5' };
```



# Initialization

Aggregate initialization

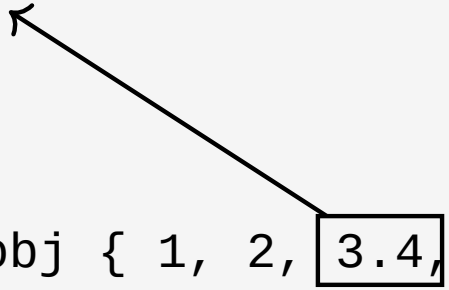
```
1  struct My_Struct
2  {
3    int a;
4    int b;
5    double c;
6    char d;
7  };
8
9  My_Struct obj { 1, 2, 3.4, '5' };
```



# Initialization

Aggregate initialization

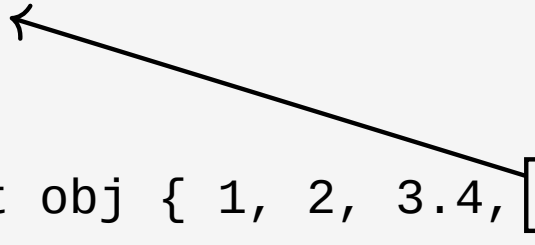
```
1  struct My_Struct
2  {
3    int a;
4    int b;
5    double c;
6    char d;
7  };
8
9  My_Struct obj { 1, 2, 3.4, '5' };
```



# Initialization

Aggregate initialization

```
1  struct My_Struct
2  {
3    int a;
4    int b;
5    double c;
6    char d;
7  };
8
9  My_Struct obj { 1, 2, 3.4, '5' };
```



# Initialization

Be careful with paranthesis in initialization

```
1 // default initialized
2 // int variable
3 int x {};
```

```
1 // function returning int
2 // taking no parameters
3 int x ();
```

# Initialization

Be careful with paranthesis in initialization

- Initialization with curly braces are recommended
- Partly because then the compiler will warn us when we have narrowing conversions
- But also because we **must** have curly braces when default-initializing a variable: parenthesis will turn the variable into a function instead which will lead to very confusing error messages.

# Initialization

What will happen?

```
1  int main()
2  {
3    int x{};
4    cout << x << " ";
5    int y = 3.5;
6    cout << y << " ";
7    int z {3.5};
8    cout << z << endl;
9  }
```

# Initialization

What will be printed?

```
1  int main()
2  {
3    int var (int());
4    cout << var << endl;
5  }
```

www.liu.se

LINKÖPING UNIVERSITY