TDDD38/726G82:
Adv. Programming in C++
Language constructs and rules I

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

NOTE: This seminar covers seemingly basic stuff like data types, functions and IO. However, it is still relevant even if you think you already grasp everything, because here we are not discussing the *how* of these things, but rather the underlying reasons behind behaviours of C++ relating to these concepts.

1 Data Types
2 Functions
3 IO

1 Data Types
2 Functions
3 IO

Data Types

What is a data type?

- A data type is an *abstraction*.
- It represents an *interpretation* of raw data.
- Formally it is a set of functions, operations and restrictions that relate to a piece of data.
- Data types are generally a tool for translating human ideas into machine executable instructions via the compiler.

Data Types

What is a data type?

These are *bits*. What do they mean?

1	0	0	1	0	0	1	1

Data Types

What is a data type?

Maybe it is an integer in base 2?

1	0	0	1	0	0	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Data Types

What is a data type?

Maybe it is an integer in base 2?

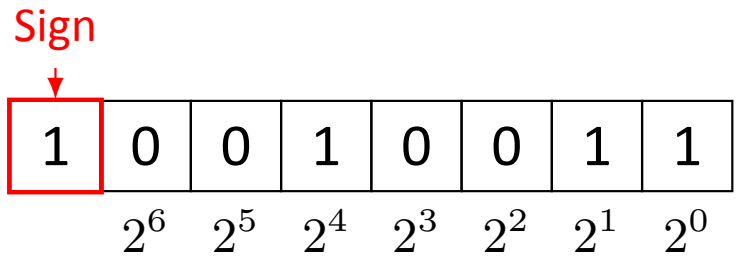
1	0	0	1	0	0	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$128 + 16 + 2 + 1 = 147$$

Data Types

What is a data type?

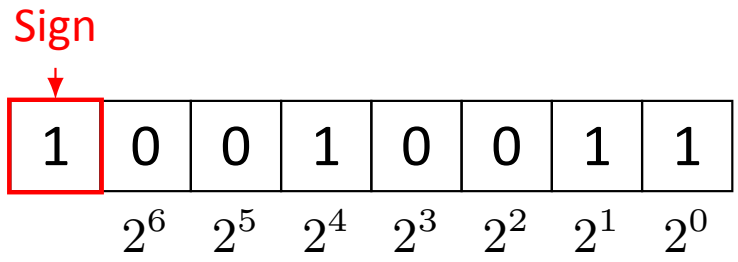
Does it have a sign bit?



Data Types

What is a data type?

Does it have a sign bit?



$$(-1)^1(16+2+1) = -19$$

Data Types

What is a data type?

Maybe in reverse?

1	0	0	1	0	0	1	1
2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7

Data Types

What is a data type?

Maybe in reverse?

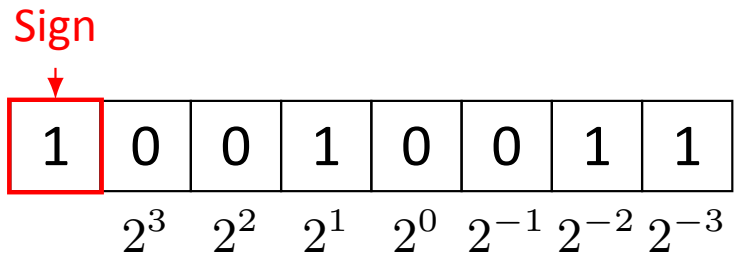
1	0	0	1	0	0	1	1
2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7

$$1 + 4 + 64 + 128 = 197$$

Data Types

What is a data type?

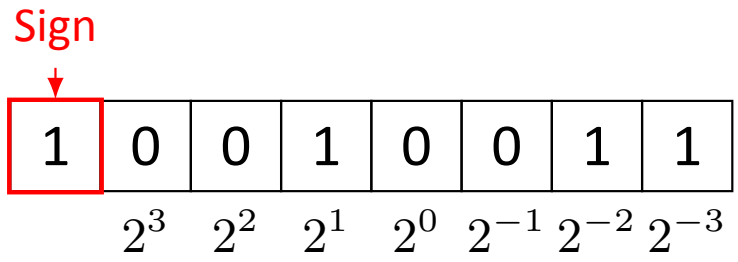
Fractional?



Data Types

What is a data type?

Fractional?

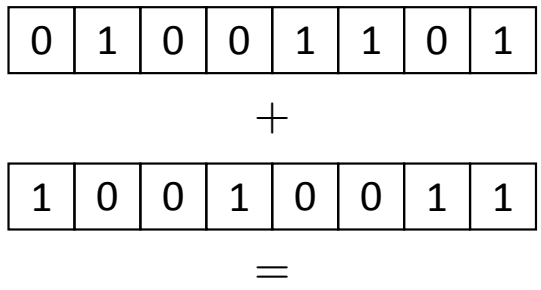


$$(-1)^1(2 + \tfrac{1}{4} + \tfrac{1}{8}) = 2.375$$

Data Types

Why does it matter?

What the operation means depends on interpretation



Data Types

Why does it matter?

Positive integers?

0	1	0	0	1	1	0	1	$= 77$

$+$

1	0	0	1	0	0	1	1	$= 147$

$=$

1	1	1	0	0	0	0	0	$= 224$

Data Types

Why does it matter?

With sign bit?

0	1	0	0	1	1	0	1	$= 77$
				$+$				
1	0	0	1	0	0	1	1	$= -19$
				$=$				
0	0	1	1	1	0	1	0	$= 58$

Data Types

Why does it matter?

Did you notice that the result differed?

0	1	0	0	1	1	0	1	$= 77$

$+$

1	0	0	1	0	0	1	1	$= -19$

$=$

0	0	1	1	1	0	1	0	$= 58$

Data Types

Data type categories

- Fundamental types

Data Types

Data type categories

- Fundamental types
  - Integer types

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters
  - Booleans

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters
  - Booleans
- Enumeration types (`enum`)

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters
  - Booleans
- Enumeration types (`enum`)
- Compound types (Covered later)

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters
  - Booleans
- Enumeration types (`enum`)
- Compound types (Covered later)
  - Pointers/reference

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters
  - Booleans
- Enumeration types (`enum`)
- Compound types (Covered later)
  - Pointers/reference
  - Arrays

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters
  - Booleans
- Enumeration types (`enum`)
- Compound types (Covered later)
  - Pointers/reference
  - Arrays
  - Aggregate (`struct`)

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters
  - Booleans
- Enumeration types (`enum`)
- Compound types (Covered later)
  - Pointers/reference
  - Arrays
  - Aggregate (`struct`)
  - Classes (`class`)

Data Types

Data type categories

- Fundamental types
  - Integer types
  - Floating point numbers
  - Characters
  - Booleans
- Enumeration types (`enum`)
- Compound types (Covered later)
  - Pointers/reference
  - Arrays
  - Aggregate (`struct`)
  - Classes (`class`)
  - Unions (`union`)

Data Types

Implementation-defined behaviours

- How many bits in a byte?

Data Types

Implementation-defined behaviours

- How many bits in a byte? ≥ 8

Data Types

Implementation-defined behaviours

- How many bits in a byte? ≥ 8
- How many bits in an `int`?

Data Types

Implementation-defined behaviours

- How many bits in a byte? ≥ 8
- How many bits in an `int`? ≥ 16

Data Types

Implementation-defined behaviours

- How many bits in a byte? ≥ 8
- How many bits in an `int`? ≥ 16
- Is there a sign bit in `int`?

Data Types

Implementation-defined behaviours

- How many bits in a byte? ≥ 8
- How many bits in an `int`? ≥ 16
- Is there a sign bit in `int`? Probably not

Data Types

Implementation-defined behaviours

- Different platforms have different properties which changes what is and isn't effective usage of memory
- Number of bits in a byte varies from system to system (mostly 8 on modern hardware though, but older systems might still vary)
- On systems with more than 8 bits per byte, it would be wasteful and inefficient to treat each byte as 8 just for simplicity.
- `int` is (informally) the most efficient integer representation
- On most modern systems 32-bit integers are the most efficient, but that is not true for *all* systems.
- There are many different ways to represent negative numbers: sign bits, 1s and 2s complement etc.

Data Types

Implementation-defined behaviours

- C++ is designed to work *and* be efficient on any imaginable platform
- Standardizing certain properties would therefore be detrimental for the overall usefulness of C++ on certain platforms
- Thus the standard lets the compiler/platform decide certain properties (such as #bits in a byte, #bits in `int`, negative number representation and many many more things)
- This means however that we should not *assume* these properties when writing platform-independent code.

Data Types

Fixed width integers

- `std::int32_t`

Data Types

Fixed width integers

- `std::int32_t`
- `std::uint32_t`

Data Types

Fixed width integers

- `std::int32_t`
- `std::uint32_t`
- `std::uint_least32_t`

Data Types

Fixed width integers

- `std::int32_t`
- `std::uint32_t`
- `std::uint_least32_t`
- `std::uint_fast32_t`

Data Types

Fixed width integers

- If you *require* a certain number of bits in your integers, you might think that `std::int32_t` (or similar types) are good: but what if the system have 36 bit bytes? Then there is no conceivable way to use 32 bits efficiently (since we would have to slice away 4 bits).
- Therefore it is better to use the type `std::int_least32_t`, since this guarantees *at least* 32 bits.
- There is also `std::int_fast32_t` which is the most efficient integer that has at least 32 bits.
- The difference between `std::int_leastN_t` and `std::int_fastN_t` is that `std::int_leastN_t` tries to be as small as possible while `std::int_fastN_t` tries to be as fast as possible.

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

Basic compound types

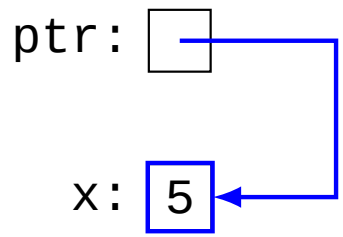
```
1  int x { 3 };
2
3
4
5
```

x: 5

Data Types

Basic compound types

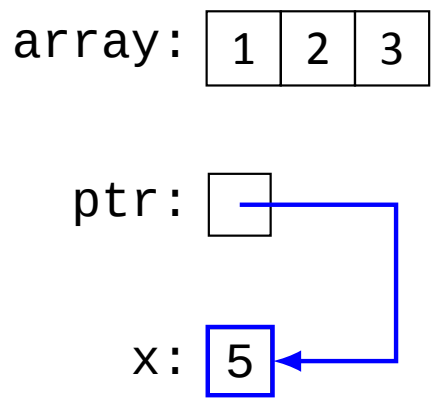
```
1 int x { 3 };
2 int* ptr { &x };
3
4
```



Data Types

Basic compound types

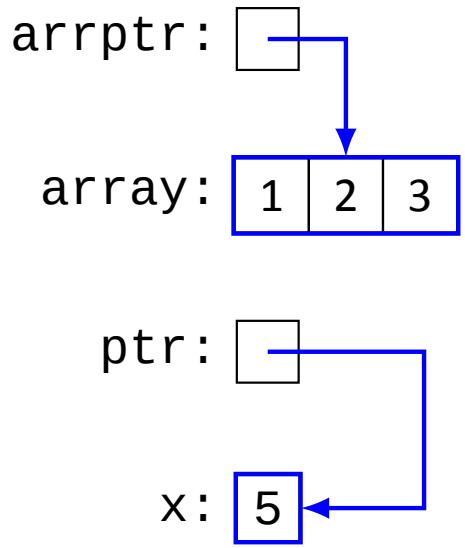
```
1 int x { 3 };
2 int* ptr { &x };
3 int array[3] { 1, 2, 3 };
4
```



Data Types

Basic compound types

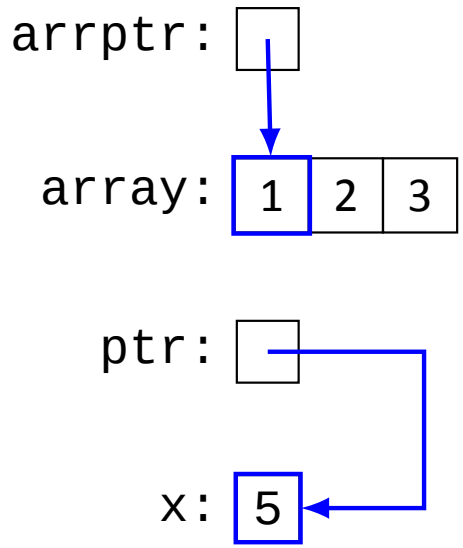
```
1 int x { 3 };
2 int* ptr { &x };
3 int array[3] { 1, 2, 3 };
4 int (*arrptr)[3] { &array };
```



Data Types

Basic compound types

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



Data Types

Basic compound types

- I am generally going to assume that you understand what a pointer is and what a fixed-size array is.
- If you don't understand pointers, look at the slides linked under this seminar on the course website.
- Here is a quick explanation of pointers: a pointer is the memory address of the first byte in some piece of data of the specified type.
- A fixed-size array is an array where the number of elements is 1) fixed (i.e. it never changes) and 2) known during compilation (meaning the size can be deduced without ever running the code).
- A pointer to an array is an address to the first byte in the first element of an array. Arrays can be passed as a normal pointer to the first element, but then we must also remember the size. With array-pointers the compiler remembers the size by embedding it into the type.

Data types

CV-qualifiers: `const`

A variable can generally be modified...

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

Data types

CV-qualifiers: `const`

But it does depend on the data type...

OK!

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

Data types

CV-qualifiers: `const`

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

Data types

CV-qualifiers: `const`

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

Compile Error!

Data types

CV-qualifiers: `const`

A variable that is `const` **cannot** be modified...

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

Data types

CV-qualifiers: `const`

> `const`ness is not a property of a variable, instead it is a property of the *data type*...

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

Data types

CV-qualifiers: `const`

Which means that `const` is a modifier that marks the type as constant (unchanging)

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

Data types

CV-qualifiers: `const`

This has the interesting side-effect that for example `int` and `int const` are two *different* data types.

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

Data types

CV-qualifiers: `volatile`

Suppose we are on an embedded system where writing to the variable `display` means sending some data over a hardware bus to some other hardware component.

```
1  int display { }; // Direct bus access
2  for (int i { 0 }; i < 10; ++i)
3  {
4    display = i;
5  }
```

Data types

CV-qualifiers: `volatile`

Then this code might cause a problem if we enable optimizations for our compilation...

```
1  int display { }; // Direct bus access
2  for (int i { 0 }; i < 10; ++i)
3  {
4    display = i;
5  }
```

Data types

CV-qualifiers: `volatile`

In particular, what will happen is that the compiler notices that we write a lot to `display`, however we never once *read* from it. So from the point-of-view of the compiler this variable is unnecessary. Meaning the compiler will completely remove *all* writes to it.

```
1 int display { }; // Direct bus access
2 for (int i { 0 }; i < 10; ++i)
3 {
4   display = i;
5 }
```

Data types

CV-qualifiers: `volatile`

This happens because the compiler is not aware of the fact that writing to `display` causes side-effects *outside* of the program (i.e. that the hardware is listening to that specific variable)

```
1  int display { }; // Direct bus access
2  for (int i { 0 }; i < 10; ++i)
3  {
4    display = i;
5  }
```

Data types

CV-qualifiers: `volatile`

But this does **NOT** happen if we add `volatile` to the variables type.

```
1  int volatile display { }; // Direct bus access
2  for (int i { 0 }; i < 10; ++i)
3  {
4    display = i;
5  }
```

Data types

CV-qualifiers: `volatile`

`volatile` tells the compiler that *writing* and *reading* values of this type have *side-effects* that the compiler is unaware of, meaning the optimizer cannot assume that the variable is unnecessary.

```
1 int volatile display { }; // Direct bus access
2 for (int i { 0 }; i < 10; ++i)
3 {
4   display = i;
5 }
```

Data types

CV-qualifiers

Both `const` and `volatile` are modifiers to data types that changes the type

CV-qualifiers = `const`-`volatile` qualifiers

Data types

CV-qualifiers

Conceptually they are different, but syntactically they work exactly the same way

CV-qualifiers = `const`-`volatile` qualifiers

Data types

CV-qualifiers

Because of this they are bundled together as CV-qualifiers

CV-qualifiers = `const`-`volatile` qualifiers

Data types

CV-qualifiers

`volatile` is **NOT** relevant for this course, so when we speak of CV-qualifiers we are mainly discussing `const`, but remember that all syntax rules for `const` also applies to `volatile`

CV-qualifiers = `const`-`volatile` 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!

1 Data Types
2 Functions
3 IO

LINKÖPING UNIVERSITY

Functions

Anatomy of functions

```
1  // signature / declaration
2  int fun(int a, int b);
3
4  // implementation / definition
5  int fun(int a, int b)
6  {
7    // ...
8  }
```

Functions

Anatomy of functions

- A function has a list of parameters and a return type.
- The *function signature* is defined by the function name, the list of parameters and the return type.
- A function can be declared by writing the signature only, and then defined later.

Functions

Function overloads

```
1  int fun(int a);                  // #1
2  int fun(int a, int b);           // #2
3  int fun(float a, int b);         // #3
4  int fun(float a);                // #4
```

which version?

```
1
2
3
4
```

Functions

Function overloads

```
1  int fun(int a);                  // #1
2  int fun(int a, int b);           // #2
3  int fun(float a, int b);         // #3
4  int fun(float a);                // #4
```

which version?

```
1  fun(0);
2
3
4
```

Functions

Function overloads

```
1 int fun(int a);                    // #1
2 int fun(int a, int b);             // #2
3 int fun(float a, int b);           // #3
4 int fun(float a);                  // #4
```

which version?

```
1 fun(0);                            // #1
2
3
4
```

Functions

Function overloads

```
1  int fun(int a);                  // #1
2  int fun(int a, int b);           // #2
3  int fun(float a, int b);         // #3
4  int fun(float a);                // #4
```

which version?

```
1  fun(0);                          // #1
2  fun(0.0f, 0);
3
4
```

Functions

Function overloads

```
1  int fun(int a);                  // #1
2  int fun(int a, int b);           // #2
3  int fun(float a, int b);         // #3
4  int fun(float a);                // #4
```

which version?

```
1  fun(0);                          // #1
2  fun(0.0f, 0);                    // #3
3
4
```

Functions

Function overloads

```
1  int fun(int a);                  // #1
2  int fun(int a, int b);           // #2
3  int fun(float a, int b);         // #3
4  int fun(float a);                // #4
```

which version?

```
1  fun(0);                          // #1
2  fun(0.0f, 0);                    // #3
3  fun(0, 0);
4
```

Functions

Function overloads

```
1  int fun(int a);                  // #1
2  int fun(int a, int b);           // #2
3  int fun(float a, int b);         // #3
4  int fun(float a);                // #4
```

which version?

```
1  fun(0);                          // #1
2  fun(0.0f, 0);                    // #3
3  fun(0, 0);                       // #2
4
```

Functions

Function overloads

```
1  int fun(int a);                  // #1
2  int fun(int a, int b);           // #2
3  int fun(float a, int b);         // #3
4  int fun(float a);                // #4
```

which version?

```
1  fun(0);                          // #1
2  fun(0.0f, 0);                    // #3
3  fun(0, 0);                       // #2
4  fun(0.0f, 0.0f);
```

Functions

Function overloads

```
1 int fun(int a);                    // #1
2 int fun(int a, int b);             // #2
3 int fun(float a, int b);           // #3
4 int fun(float a);                  // #4
```

which version?

```
1 fun(0);                            // #1
2 fun(0.0f, 0);                      // #3
3 fun(0, 0);                         // #2
4 fun(0.0f, 0.0f);                   // #3
```

Functions

Function overloads

- C++ allows for many functions with the same name, as long as the parameter list differs. Specifically the number of parameters and/or the types of the parameters must be different between each overload.
- When a function is called, the compiler retrieves a list of all functions with corresponding name, and then it compares the arguments passed to the function call with the declared parameters of the function signature.
- If the compiler finds an overload that matches the function call exactly then it simply calls that function.
- But there might be a situation (as seen is the example) where the number of parameters matches an overload, but the types does not.

Functions

Function overloads

- How should the language/compiler handle this?
- One idea is to simply forbid calling functions that doesn't match exactly.
- This idea does have some major drawbacks. Suppose that we have a function called `fun()` with signature `void fun(unsigned x)`, and we call it as `fun(0)` then this wouldn't compile since `0` has data type `int` which doesn't match the parameter exactly.
- Another, worse example, would be if the function has signature `void fun(int const x)` then `fun(0)` also wouldn't work, since `int` and `int const` are different (highly compatible) data types.

Functions

Function overloads

- We could of course make special exceptions for `const` types and different integer types, but then we run into similar problems for other types.
- A string literal, e.g. `"a string"` has data type `char const*`, so we cannot pass it to a function that takes a `std::string` etc.
- C++ has a guiding principle which states that no types should have hidden favors from the compiler. Therefore we cannot allow for these types of special cases for specific types determined by the compiler.
- So the only reasonable thing to do here is to allow calling functions with slight variations in the parameter data types.

Functions

Function overloads

- So what should we do in that case?
- Well, if a value of type A *can* be converted into a value of type B, then that should be allowed.
- However, the compiler should only do this if no exact match exists.
- The final call in the previous code example matches both overload #2 and #3, which one should it pick?
- We could of course forbid this, but then the previous problems discussed arises again.
- Therefore the language instead states that in those cases it should match the overload that requires the *fewest* conversions.

Functions

Calling overloads

1. Exact match?
2. Convert one parameter?
3. Convert two parameters?
4. ...

Functions

Implicit conversion

- Promotion
- Narrowing
- Numeric

Functions

Implicit conversion

- The compiler is allowed to do a *promotion* of a data type, meaning convert it into the same category of values but with a bigger value range. For example converting `short int` to `int`, or `int` to `long int`. Note that `float` to `double` would also be a promotion.
- *Narrowing* conversions are often permitted, but commonly the compiler produces a warning if that happens. A *narrowing* conversion is the opposite of a *promotion* (reducing the value range).
- It is also permitted to convert between integers and floating point numbers. This is called *numeric* conversions.

Functions

Circular dependent functions: Problem

```
1
2
3
4  bool even(unsigned value)
5  {
6    if (value == 0) return true;
7    return odd(value - 1);
8  }
9
10 bool odd(unsigned value)
11 {
12   if (value == 0) return false;
13   return even(value - 1);
14 }
```

Functions

Circular dependent functions: Solution

```
1  bool even(unsigned value);
2  bool odd(unsigned value);
3
4  bool even(unsigned value)
5  {
6    if (value == 0) return true;
7    return odd(value - 1);
8  }
9
10 bool odd(unsigned value)
11 {
12   if (value == 0) return false;
13   return even(value - 1);
14 }
```

Functions

Circular dependent functions

- C++ is designed to work with a one-pass compiler, i.e. it should be possible to process and compile a piece of code once going line-by-line.
- A one-pass compiler could not possibly figure out which function to call at line 7, since it hasn't yet seen the signature of `odd()`.
- One possible design of this would be to allow the compiler to generate a function signature based on the surrounding context.

Functions

Circular dependent functions

- Figuring out the return type isn't always possible from context, so how should that be handled?
- Any typos in function names would silently produce a new function signature instead of producing an error directly, which would – potentially – lead to harder to understand errors.
- How do we handle the case when we are calling a non-existing function overload? Should it produce a new overload which potentially isn't implemented? Should it perform implicit type conversions to match an existing overload? This means that we would have to define when it is allowed to use a previously defined function with mismatching parameters types and when a new signature should be produced. This runs the risk of producing non-intuitive behaviours.

Functions

Circular dependent functions

- But this runs a different problem, which is seen in the previous code example: if there are two functions who depend on each other we cannot possibly define them both before calling them.
- The solution is quite simple: allow the user to *declare* the functions before defining them, thus telling the compiler all the necessary information for implementing a function call.

1 Data Types
2 Functions
3 IO

IO

Basic IO stuff

```
1  #include <iostream>
2
3
4  int main()
5  {
6    int x;
7    int y;
8
9    std::cin >> x >> y;
10   std::cout << "x + y = " << x + y << std::endl;
11 }
```

IO

But why?

- C uses functions `printf()` and `scanf()` which takes, as a first parameter, a so-called *format* string, and the remaining parameters are printed based on the format string. For example:
  ```
  printf("x + y = %d\n", x + y);
  ```
  where %d specifies that the *first* argument (in this case) is printed as a decimal (base-10) integer.
- This style is fairly simple but requires you, the programmer, to know the string specification for each parameter you pass. How do you print a string? You use %s. How do you print an integer as a hexadecimal? %x. How do you print an unsigned integer in base 10? %u, and so on.
- The function also *blindly* trusts your specified type. If you say you want to print a string, but then pass an integer it is going to try to print the integer bits as if they were a string. This will probably cause huge issues.

IO

But why?

- This is especially egregious since the *data types* are supposed to handle this properly as part of the language design.
- In essence what this design we ignore the data types entirely and let the user act as to how different data types are printed.
- This will cause major technical debt if we for example need to change the type of some variables.

IO

But why?

- C++ introduced the concept of function overloading.
- With this a new design could be implemented which was driven by the types of the variables rather than what the user specified.
- It would for example be designed something like this:

```
1        write(std::cout, "x + y = ");
2        write(std::cout, x + y);
3        write(std::cout, "\n");
```

Where there is an overload of the `write()` function for different types.

IO

But why?

- However, this is quite cluttered and there is a lot of boilerplate code having to be written. The dream is to be able to put everything on one line.
- In modern C++ this can easily be solved using techniques from C++11 and onwards. But the IO library was written way before those features.
- Because of this, the compromise was to use *operator overloading* (which we will talk about in the next seminar) to make a usable interface that was also *type safe* (i.e. how things are printed are based on the data type).
- In C++23 they finally began working on fixing and modernizing the IO features of C++ (see next slide).
- There are however other benefits with the old way compared to the modern alternatives, but these benefits will be revealed later on.

IO

C++23

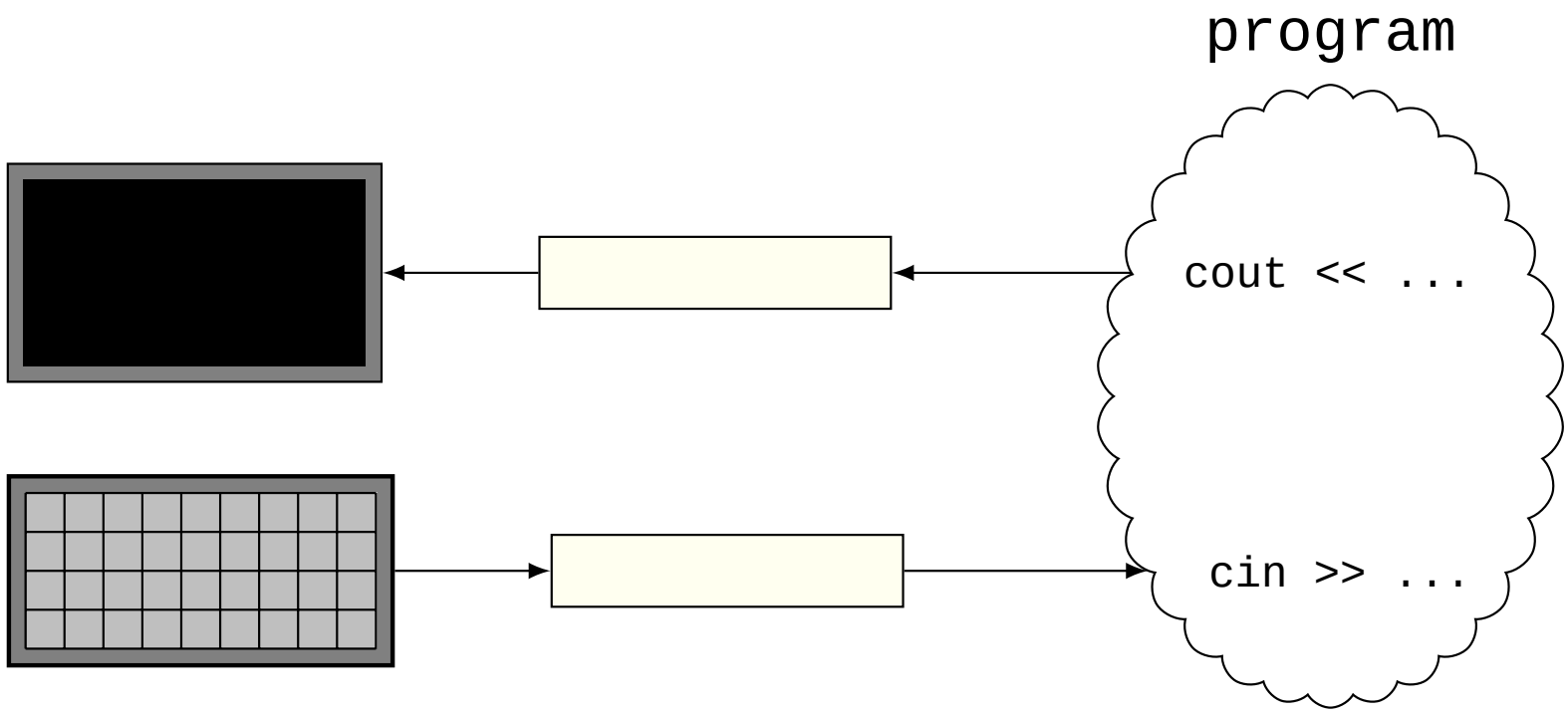
```cpp
#include <iostream>
#include <print>

int main()
{
  int x;
  int y;

  std::cin >> x >> y;
  std::println("x + y = {}", x + y);
}
```

IO

Buffered



IO

Buffered

- One important aspect of the C++ streams is that they are *buffered*
- This means that they do not read or write directly from the terminal
- Instead they store data in an intermediate buffer which are *flushed* at appropriate times
- This is done to reduce the number of context switches needed

www.liu.se

LINKÖPING UNIVERSITY