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

Fundamentals

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

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

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

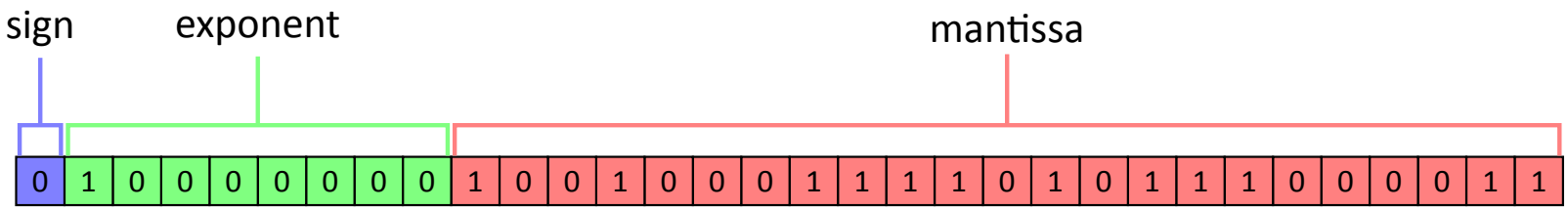
Data types

Floating-point types

- `float`
- `double`
- `long double`

Data types

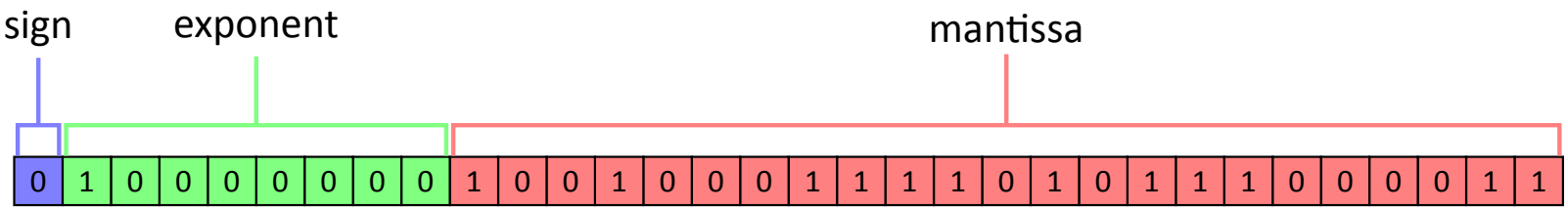
Floating-point types



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

Data types

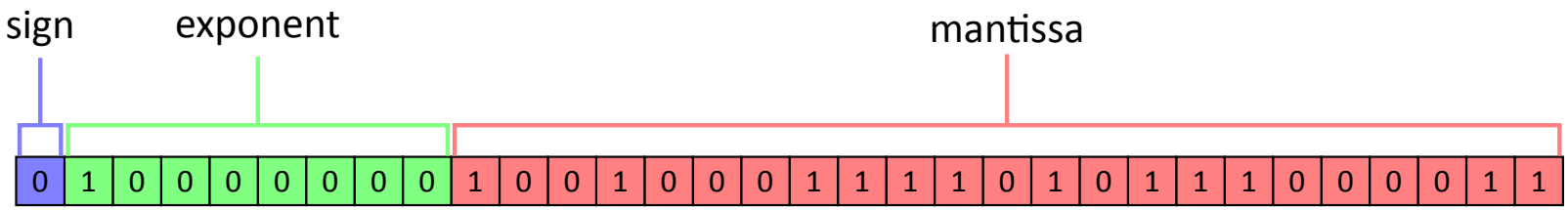
Floating-point types



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

Data types

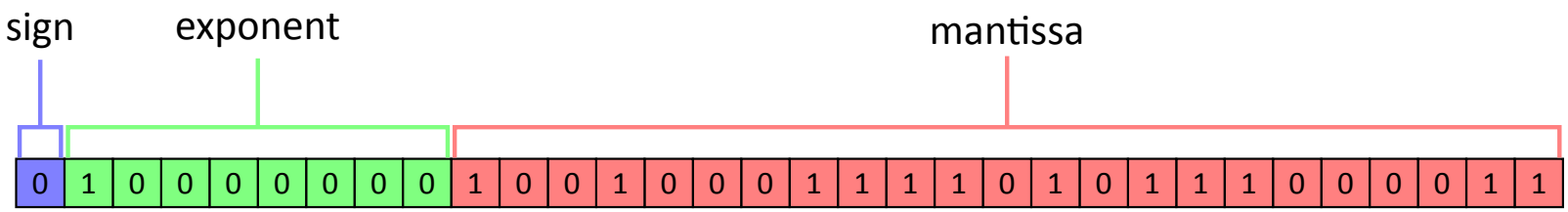
Floating-point types



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

Data types

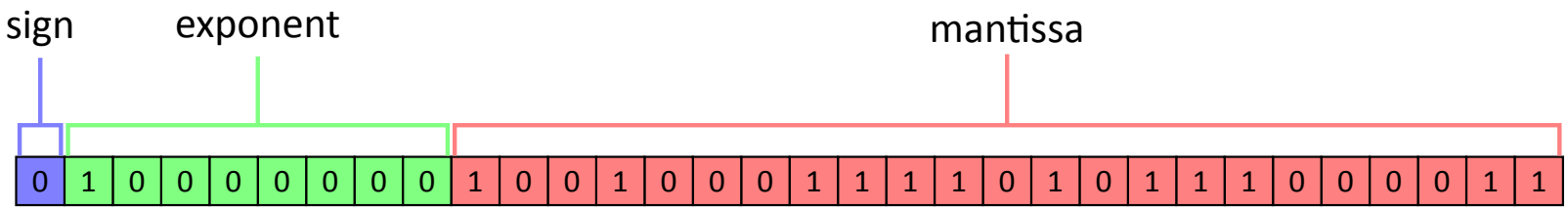
Floating-point types



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

Data types

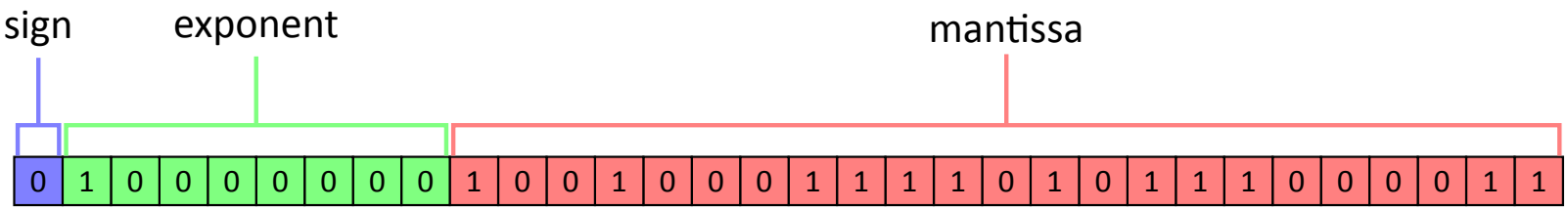
Floating-point types



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

Data types

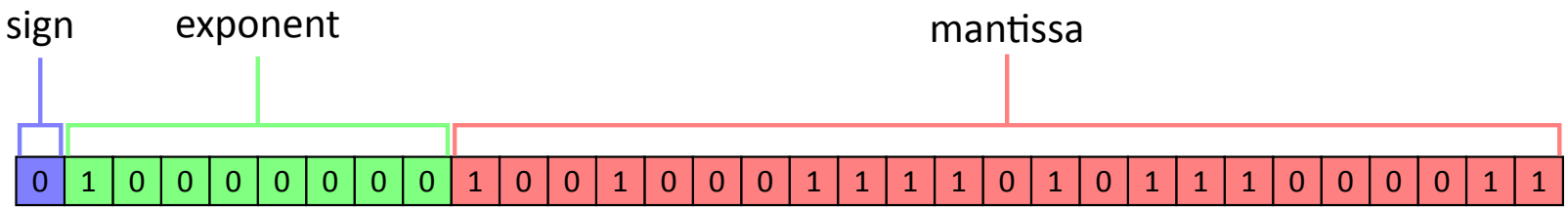
Floating-point types



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

Data types

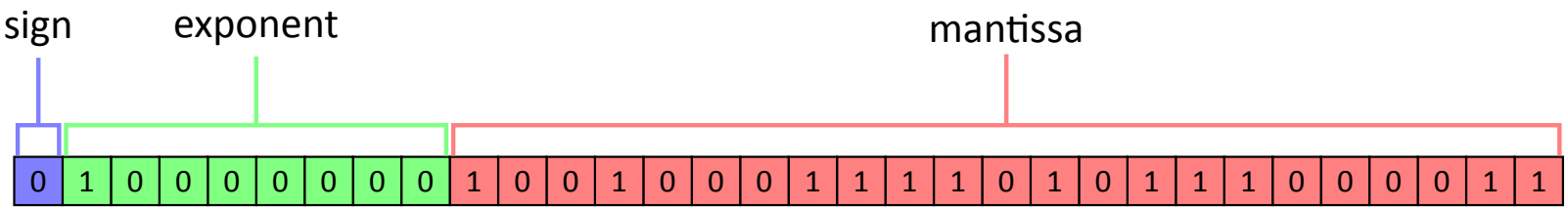
Floating-point types



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

Data types

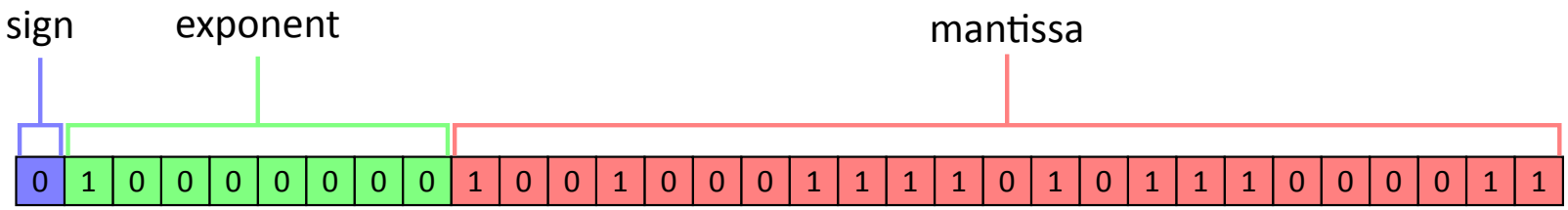
Floating-point types



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

Data types

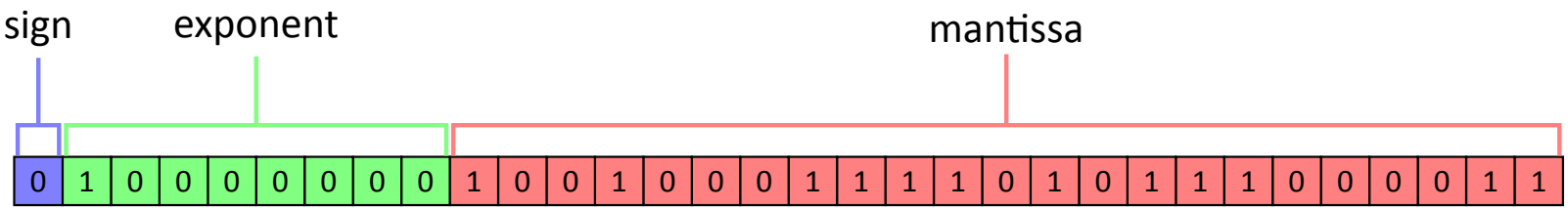
Floating-point types



$$2 \cdot (1.10010001111010111000011_2)$$

Data types

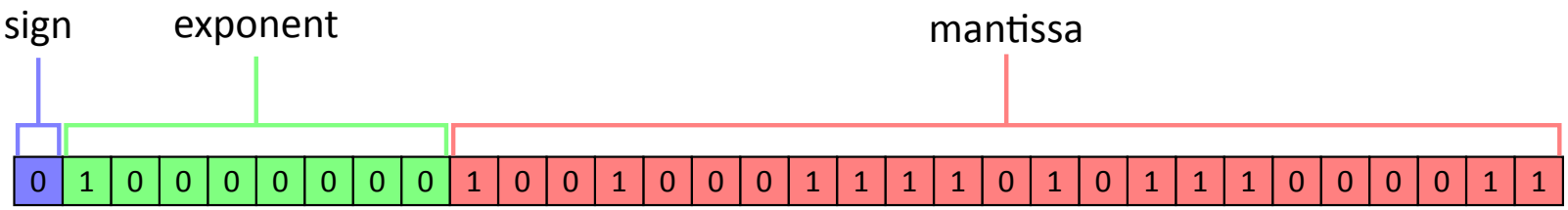
Floating-point types



$$2 \cdot 1.5700000524520874$$

Data types

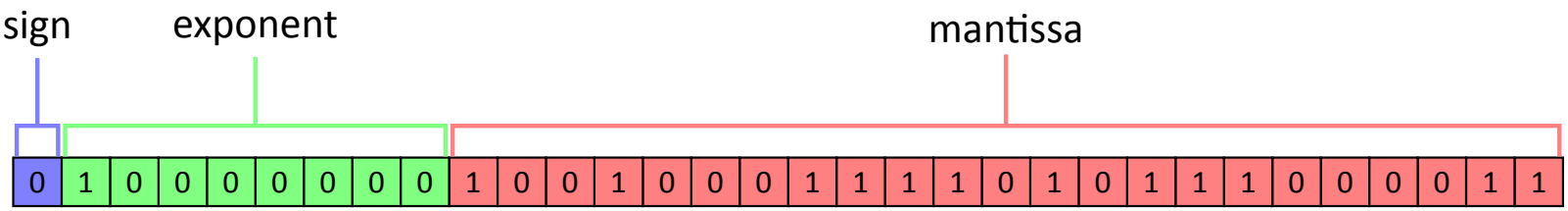
Floating-point types



$$2 \cdot 1.5700000524520874$$

Data types

Floating-point types



$$\approx 3.14$$

Data types

Fundamental data types

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

Data types

Other fundamental types

- `bool`
- `void`
- `std::nullptr_t`

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 example

```
int array[3] { 1, 2, 3 };

array[0] = 2;
array[2] = array[2] - 1;

for (unsigned i { 0 }; i < 3; ++i)
{
  std::cout << array[i] << std::endl;
}
```

Data types

Array types example

```
int array[3] { 1, 2, 3 };

array[0] = 2;
array[2] = array[2] - 1;

for (unsigned i { 0 }; i < 3; ++i)
{
  std::cout << array[i] << std::endl;
}
```

Data types

Array types example

```
int size { };
std::cout << "Enter size: ";
std::cin >> size;

int array[size] { };
```

Data types

Array types example

```
int size { };
std::cout << "Enter size: ";
std::cin >> size;

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] { };
```

Forbidden!

Data types

Data type categories

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

Data types

Enumeration types

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

Data types

Enumeration types

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

```
Direction dir { NORTH };
switch (dir)
{
  case NORTH: /* ... */ break;
  case EAST:  /* ... */ break;
  case SOUTH: /* ... */ break;
  case WEST:  /* ... */ break;
}
```

Data types

Enumeration types

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

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

Data types

Enumeration types

```
enum Log_Level : char
{

  DEBUG   = 0b0001, // 1
  INFO    = 0b0010, // 2
  WARNING = 0b0100, // 4
  ERROR   = 0b1000  // 8

};
```

Data types

Enumeration types

```
enum Log_Level : char
{

  DEBUG   = 0b0001, // 1
  INFO    = 0b0010, // 2
  WARNING = 0b0100, // 4
  ERROR   = 0b1000  // 8

};
```

```
Log_Level active {
  INFO | WARNING | ERROR
};

if (active & DEBUG)
  // write DEBUG logs
else if (active & INFO)
  // write INFO logs
// ...
```

Data types

Enumeration types

```
enum Log_Level : char
{

  DEBUG   = 0b0001, // 1
  INFO    = 0b0010, // 2
  WARNING = 0b0100, // 4
  ERROR   = 0b1000  // 8

};
```

```
char active {
 0b0010 | 0b0100 | 0b1000
};

if (active & 0b0001)
  // write DEBUG logs
else if (active & 0b0010)
  // write INFO logs
// ...
```

Data types

A problem with enumeration types

```
enum Log_Level : char
{

  DEBUG   = 0b0001, // 1
  INFO    = 0b0010, // 2
  WARNING = 0b0100, // 4
  ERROR   = 0b1000  // 8

};
```

```
enum Status
{

  PENDING,   //  0
  ACCEPTED,  //  1
  DENIED,    //  2
  ERROR = -1 // -1

};
```

Data types

A problem with enumeration types

```
enum Log_Level : char
{

  DEBUG   = 0b0001, // 1
  INFO    = 0b0010, // 2
  WARNING = 0b0100, // 4
  ERROR   = 0b1000  // 8

};
```

```
enum Status
{

  PENDING,   //  0
  ACCEPTED,  //  1
  DENIED,    //  2
  ERROR = -1 // -1

};
```

Data types

A problem with enumeration types

```
enum Log_Level : char
{

  DEBUG   = 0b0001, // 1
  INFO    = 0b0010, // 2
  WARNING = 0b0100, // 4
  ERROR   = 0b1000  // 8

};
```

```
enum Status
{

  PENDING,   //  0
  ACCEPTED,  //  1
  DENIED,    //  2
  ERROR = -1 // -1

};
```

Ambiguous!

Data types

Scoped enums

```
enum class Status
{

  PENDING,
  ACCEPTED,
  DENIED,
  ERROR = -1

};
```

Data types

Scoped enums

```
enum struct Status
{

  PENDING,
  ACCEPTED,
  DENIED,
  ERROR = -1

};
```

Data types

Scoped enums

```
enum struct Status
{

  PENDING,
  ACCEPTED,
  DENIED,
  ERROR = -1

};
```

```
// PENDING is default
Status status { };
while (status == Status::PENDING)
{
  status = handle();
  if (status == Status::DENIED)
    // ...
  // ...
}
```

Data types

Scoped enums

```
enum struct Status
{

  PENDING,
  ACCEPTED,
  DENIED,
  ERROR = -1

};
```

```

// Not ambiguous since Status
// is a scoped enum!

Status status { Status::ERROR };

Log_Level level { ERROR };


```

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!

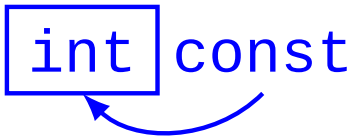
Data types

CV-qualifiers

```
int const
```

Data types

CV-qualifiers



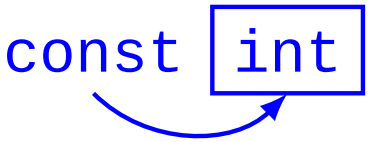
Data types

CV-qualifiers

```
const int
```

Data types

CV-qualifiers



Data types

CV-qualifiers

```
int const * const
```

Data types

CV-qualifiers

```
int const * const
```

Data types

CV-qualifiers

Data types

CV-qualifiers

```
const int * const
```

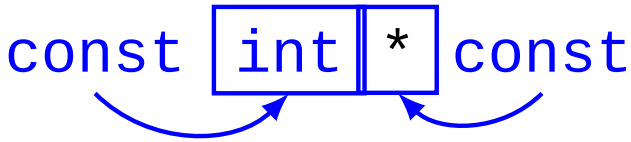
Data types

CV-qualifiers

```
const int * const
```

Data types

CV-qualifiers



Data types

C-strings

```
/* 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

LINKÖPING UNIVERSITY

Functions

Function definition

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

Functions

Multiple overloads!

```
int add(int a, int b)
{
  return a + b;
}

double add(double a, double b)
{
  return a + b;
}

int add(int a, int b, int c)
{
  return a + b + c;
}
```

Functions

Declaration & Definition

```
int add(int a, int b)
{
  if (b < 0)
    return sub(a, -b);
  return a + b;
}

int sub(int a, int b)
{
  if (b < 0)
    return add(a, -b);
  return a - b;
}
```

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 }
```

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 }
```

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 }
```

Compile Error!

Functions

Declaration & Definition

```
int sub(int a, int b);
int add(int a, int b);

int add(int a, int b)
{
  if (b < 0)
    return sub(a, -b);
  return a + b;
}

int sub(int a, int b)
{
  if (b < 0)
    return add(a, -b);
  return a - b;
}
```

Functions

Overload resolution

```
// Suppose we have:
int print(int x) { /* ... */ }

int main()
{
  print(3);    // works
}
```

Functions

Overload resolution

```
// Suppose we have:
int print(int x) { /* ... */ }

int main()
{
  print(3.0);  // works?
}
```

Functions

Overload resolution

```
// Suppose we have:
int print(int x) { /* ... */ }

int main()
{
  print(true); // works?!
}
```

LINKÖPING UNIVERSITY

Conversions

Implict conversion

- Arguments
- Operands
- Initializations
- Conditions

Conversions

Implicit conversions

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

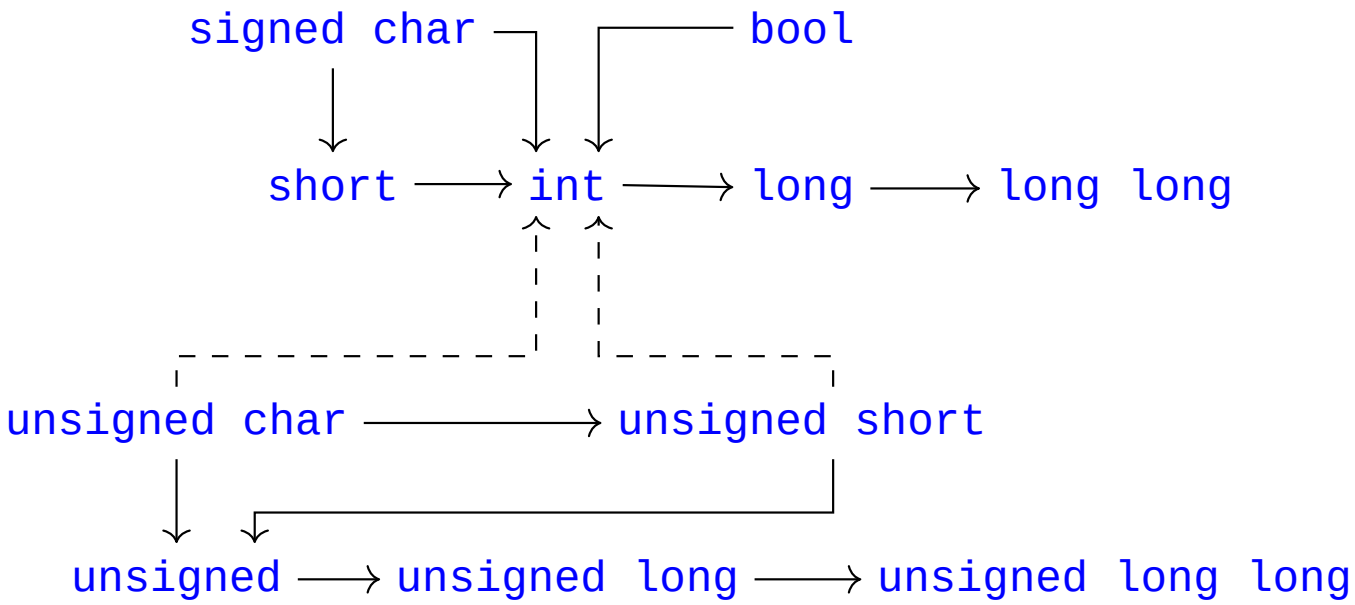
Conversions

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



Conversions

Floating-point promotions

$$\texttt{float} \longrightarrow \texttt{double} \longrightarrow \texttt{long double}$$

Conversions

Implicit conversions

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

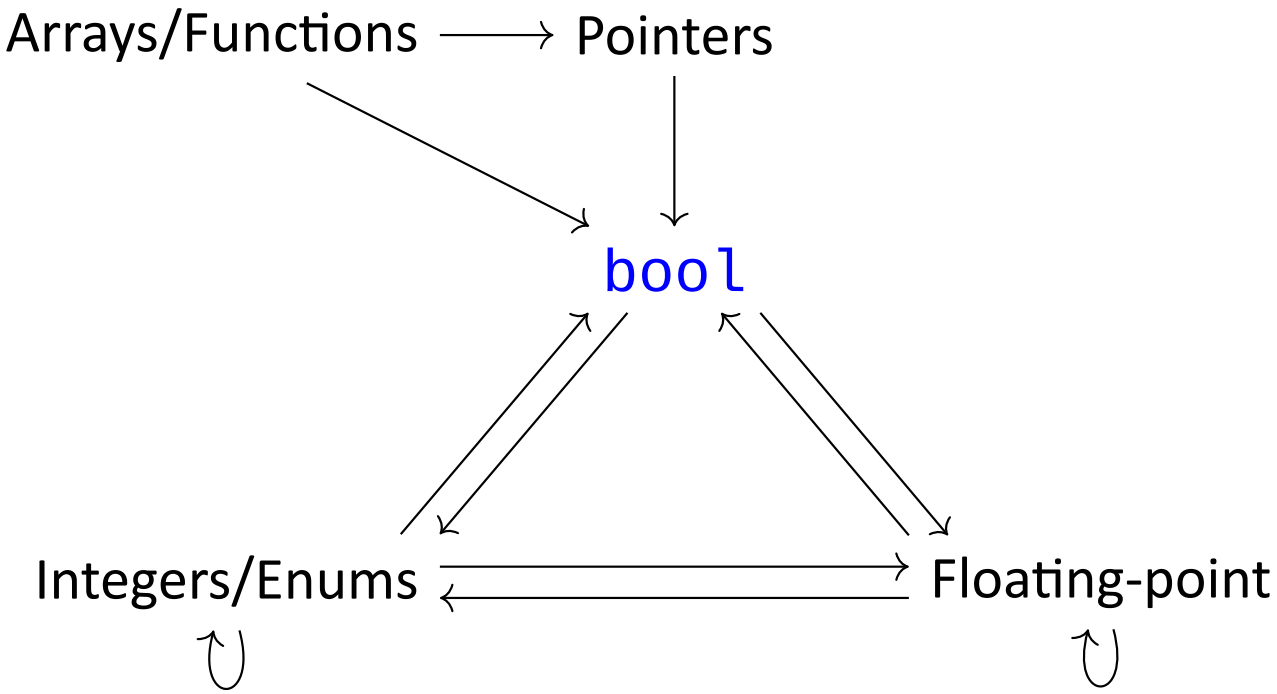
Conversions

Implicit conversions

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

Conversions

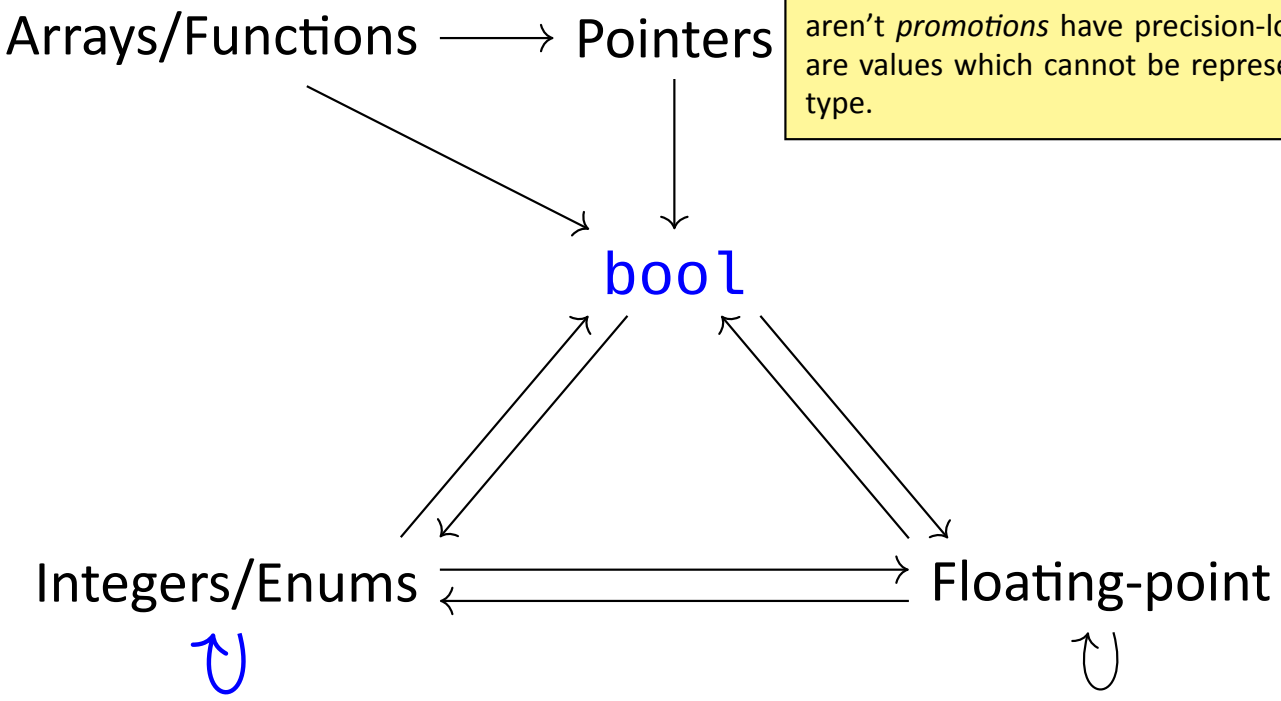
Numeric & Boolean conversions



Conversions

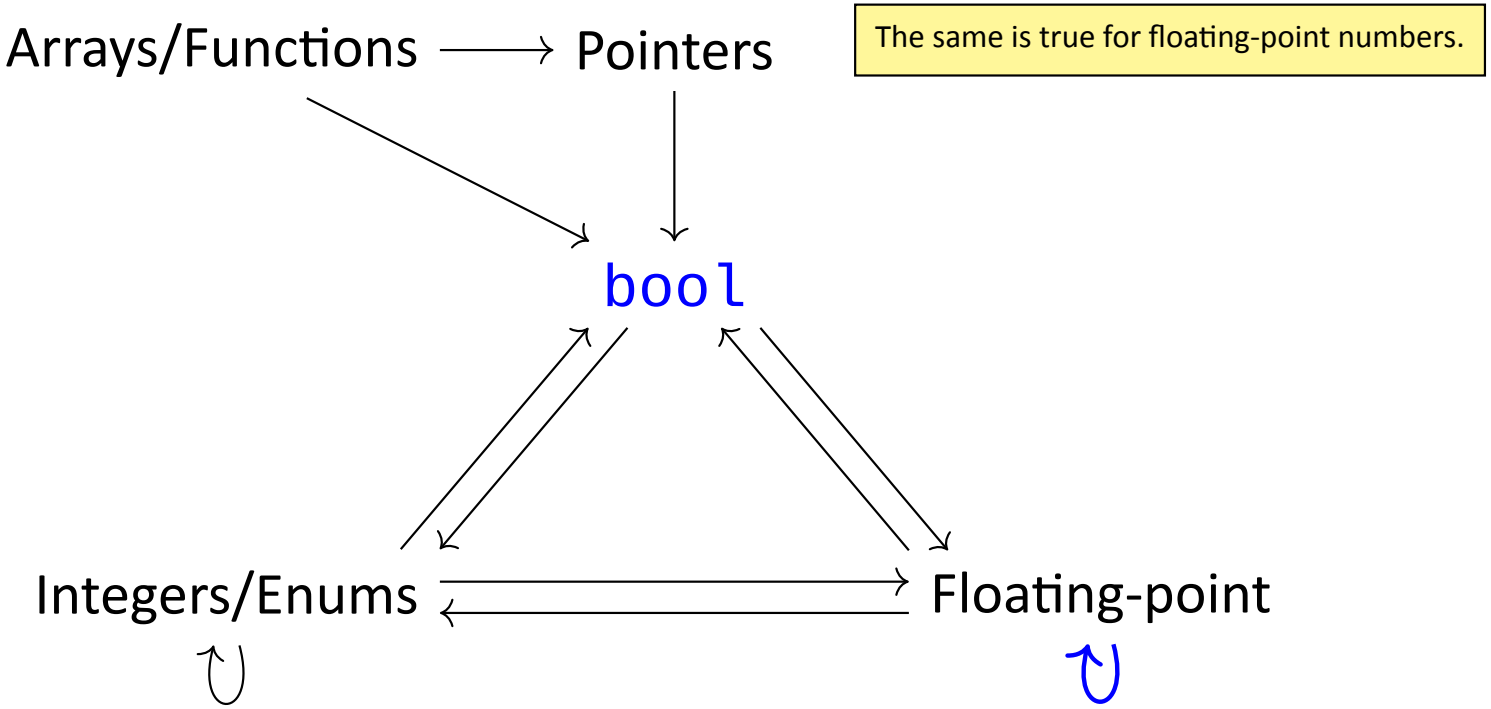
Numeric & Boolean conversions

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

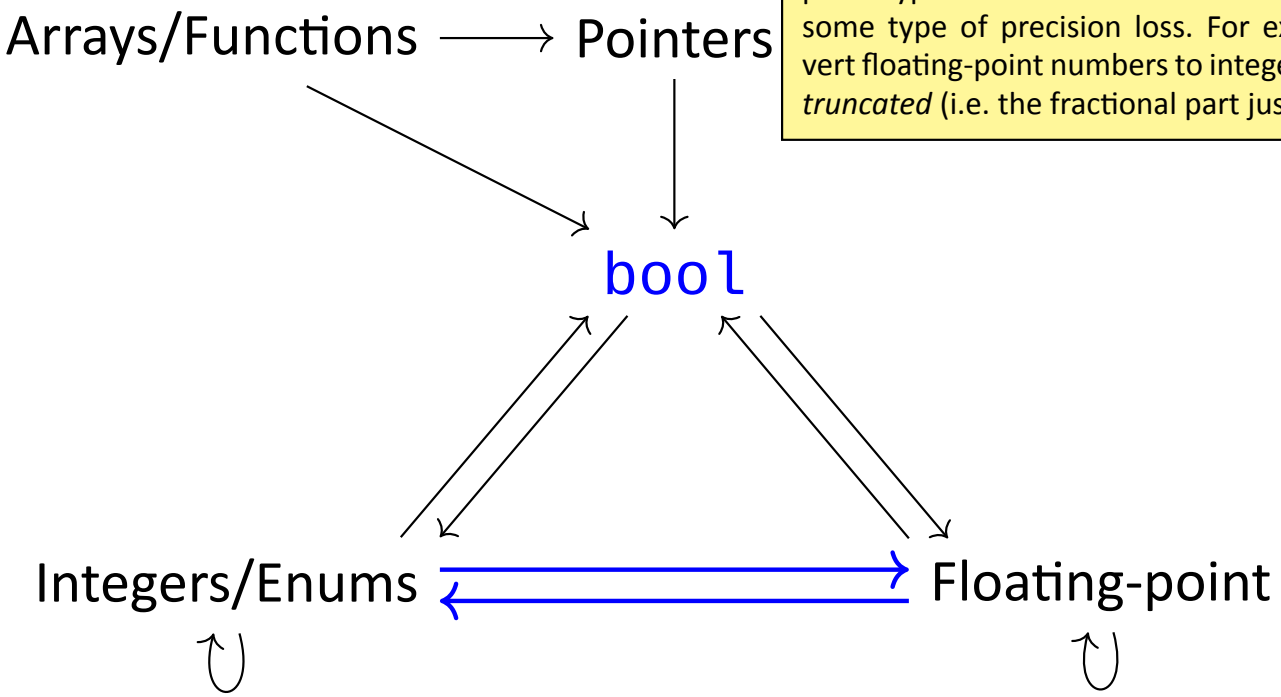
Numeric & Boolean conversions



The same is true for floating-point numbers.

Conversions

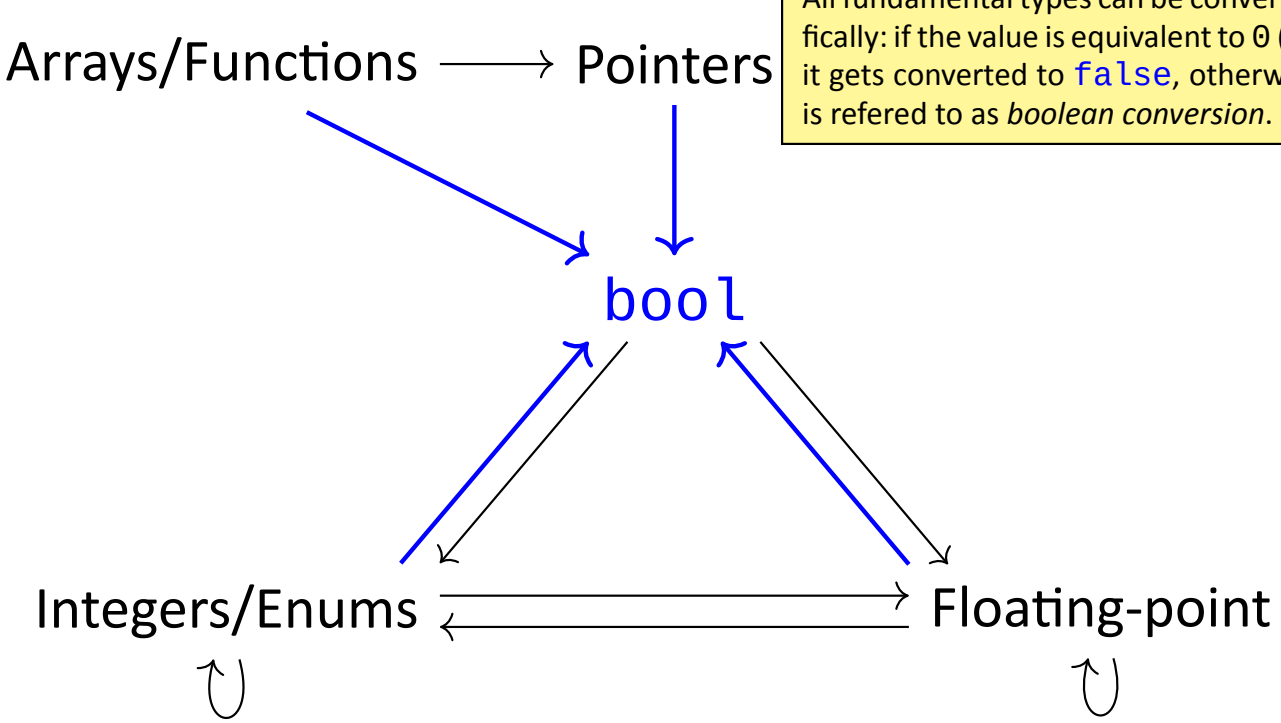
Numeric & Boolean conversions



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

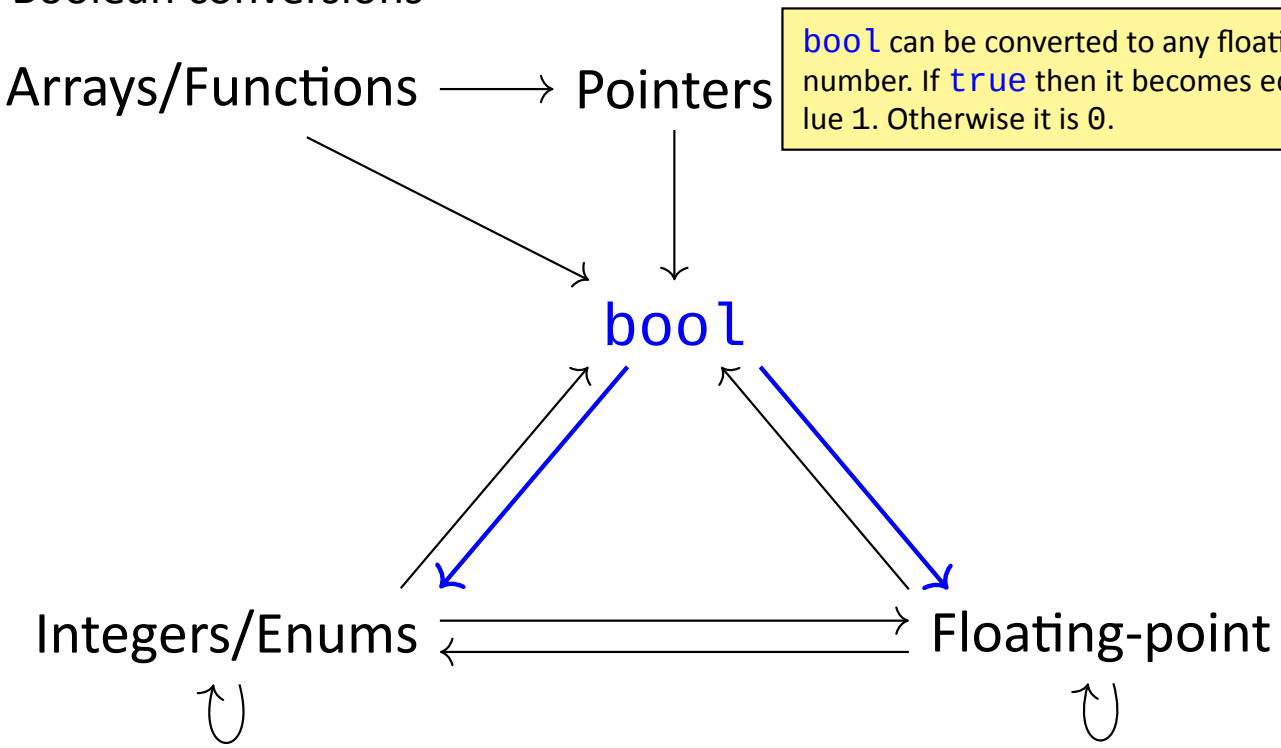
Numeric & Boolean conversions



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

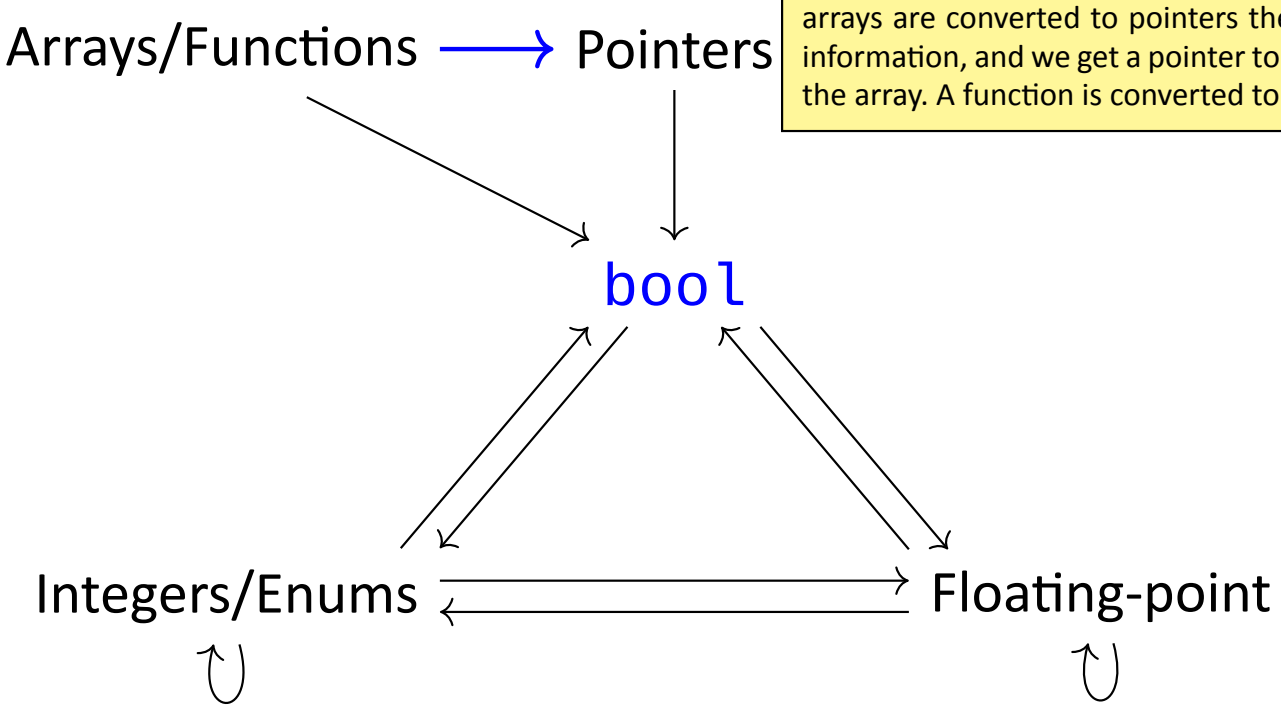
Numeric & Boolean conversions



`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

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 }
```

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

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

```
// default initialized
// int variable
int x {};
```

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

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