TDD38/726G82 - Advanced programming in C++

Basic C++

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

Initial example

What will be printed? Why?

```
#include <iostream>
using std::cout;

int main()
{
  int x { 2 };
  if (x = 0)
    cout << "x is zero\n";
  else
    cout << "Value of x: " << x << std::endl;
  return 0;
}
```

Initial example

Why?

- The condition contains an assignment
- `x` gets assigned the value `0`
- assignment returns a reference to `x`
- `x` is `0` which is convertible to `false`
- conditions in if-statements are only valid if the expression is convertible to `bool`

1 Data types
2 Functions
3 Initialization
4 Value categories
5 Conversions
6 Memory & pointers (Bonus)
7 Command-Line Arguments (Bonus)

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Data types

Type classifications

There are five classifications of data types:

- Fundamental types
- Array types
- Class types
- Enum types
- Reference/pointer types

Data types

Fundamental types

- Integer types (`int`, `unsigned int`, `short`, etc.)

Data types

Fundamental types

- Integer types (`int`, `unsigned int`, `short`, etc.)
- Floating point types (`float`, `double`, `long double`)

Data types

Fundamental types

- Integer types (`int`, `unsigned int`, `short`, etc.)
- Floating point types (`float`, `double`, `long double`)
- Character types (`char`, `wchar_t`, `unsigned char`, etc.)

Data types

Fundamental types

- Integer types (`int`, `unsigned int`, `short`, etc.)
- Floating point types (`float`, `double`, `long double`)
- Character types (`char`, `wchar_t`, `unsigned char`, etc.)
- Other types: `bool`, `void`

Data types

Fundamental types

- Types that can be used directly
- basic building blocks of all other types
- Commonly used for arithmetic operations

Data types

Array types

```
// size = 3, type = float
float farr[3] { 1.2f, 2.3f, 3.4f };

// size = 4, type = char
char carr[] { 'a', 'b', 'c', '\0' };

// size = 4, type = char
char str[] { "abc" };

// prints 'c' (0 is the first element)
std::cout << carr[2] << std::endl;
```

Data types

Array types

- Arrays of a single type
- Used for storing a fixed count of values
- The size must be known by the compiler during compilation
- The size can be set manually or deduced by the compiler
- There are better alternatives in modern C++

Data types

Array types

Some extra notes:

- `3.5` is of type `double`, while `3.5f` is of type `float`
- String literals (i.e. `"abc"`) are `char`-arrays that ends with the special character `'\0'`
- This means that `carr` and `str` are actually the same thing
- You can access individual elements by index (starting at 0), like this: `carr[2]` (retrieves the third element)

Data types

Array pointers

```
int arr1[] { 1, 2, 3 };
int arr2[] { 4, 5, 6 };

int (*ptr)[3]; // pointer to int-array of size 3

ptr = arr1;
// print 2
std::cout << ptr[1] << std::endl;

ptr = arr2;
// print 5
std::cout << ptr[1] << std::endl;
```

Data types

Array pointers

- A pointer contains a *memory address*
- It also specifies what type the value it points to have
- Normally we can have pointers to variables, like this:

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

// print value of (i.e. 5)
std::cout << *ptr << std::endl;
```

Data types

Array pointers

- But we can also have pointers to arrays:
```
int (*ptr)[3];
```
- This will simply contain the address of the first element
- However C++ will know that it is an array of the specified size since we told it that
- We can use array pointers just like normal arrays, but with one added feature: we can change which array it points to.

Data types

Class types

- `struct`, `class`
- `union`

Data types

Class types

- `struct`, `class`

```
struct Person
{
  string name; // class type
  int age; // fundamental type
};
```

- `union`

Data types

Class types

- `struct`, `class`
- `union`

```
union JSON
{
    double val;
    char const* str; // pointer type
};
```

Data types

Class types

- Types composed of several different types (called *fields* or *data members*)
- Can contain functions (called *member functions*)
- `struct` and `class` have *fields* that are set at the same time
- `union` have *fields*, but only one of them can be set at a time (they share the same memory)

Data types

Enum types

```
enum Status  // unscoped
{
  ERROR,
  PENDING,
  GRANTED = 10,
  DENIED
};



Status status { ERROR };
```

Data types

Enum types

```
enum Status : char // unscoped
{
  ERROR,
  PENDING,
  GRANTED = 10,
  DENIED
};



Status status { ERROR };
```

Data types

Enum types

```
enum Status : char // unscoped
{
  ERROR,
  PENDING,
  GRANTED = 10,
  DENIED
};

enum Flags { GOOD, ERROR };

Status status { ERROR }; // Which one?
```

Data types

Enum types

```
enum class Status : char // scoped
{
  ERROR,
  PENDING,
  GRANTED = 10,
  DENIED
};

enum Flags { GOOD, ERROR };

Status status { Status::ERROR }; // scoped value
```

Data types

Enum types

- A predefined set of discrete values
- Each possible value has a name
- Is an *integral* type
- There are two types of enums: scoped and unscoped

Data types

Enum types

- Enums are implemented by the compiler as integers
- Usually implemented as `int`, but can be specified by the user (for example `enum Status : char`)
- Each named value gets assigned to a specific integer value (first one is by-default `0`)
- Each value is represented by the previous value + 1 if not explicitly specified (see `GRANTED = 10`)

Data types

Unscoped enums

- Unscoped enums are the “normal” kind
- Each value is a global constant meaning that `ERROR` will clash since both `Status` and `Flags` contain a value of that name (notice that thay also have two different integer representations).
- So if you are using unscoped enums you have to be careful with the naming.

Data types

Scoped enums

- Scoped enums forces each named value to be directly associated with the enum itself. A enum is scoped if the `enum` keyword is followed by `struct` or `class`, like this: `enum class Status`
- This means that if we want to refer to a value from for example `Status` we have to add `Status` as a prefix, like this: `Status::ERROR`
- This is a much safer and easier way to deal with enums since we now clearly communicate what we are doing.

LINKÖPING UNIVERSITY

Functions

- Function definition
- Function declaration
- Function overload

Functions

- Function definition

```
int foo(int parameter)
{
  return parameter;
}
```

- Function declaration
- Function overload

Functions

- Function definition
- Function declaration

```
int foo(int parameter);

int foo(int parameter)
{
  return parameter;
}
```

- Function overload

Functions

- Function definition
- Function declaration
- Function overload

```
int foo(int parameter)
{
  return parameter;
}

double foo(double parameter)
{
  return parameter;
}
```

Functions

- Function definition
- Function declaration
- Function overload

```
int foo(int parameter)
{
  return parameter;
}

double foo(double a, double b)
{
  return a + b;
}
```

Functions

Why would we separate declaration and definition?

```
void foo(int x)
{
  if (x == 0)
    bar(x);
}

void bar(int x)
{
  if (x != 0)
    foo(x);
}
```

Functions

Why would we separate declaration and definition?

```
void foo(int x)
{
  if (x == 0)
    bar(x);
}

void bar(int x)
{
  if (x != 0)
    foo(x);
}
```

Compile error!

Functions

Why would we separate declaration and definition?

- The previous example won't compile:
- C++ is a *single pass* compiled language, meaning the compiler will process the code from top to bottom *once*.
- This means that when compiling `foo` the compiler finds a call to `bar` which has not yet been defined, so the compiler doesn't know what to do.
- We could solve this by defining `bar` first, but then we would get the same problem with the compiler not knowing what `foo` is.

Functions

Why would we separate declaration and definition?

```
void bar(int x); // forward declaration

void foo(int x)
{
  if (x == 0)
    bar(x);
}

void bar(int x)
{
  if (x != 0)
    foo(x);
}
```

Functions

Why would we separate declaration and definition?

- If we declare `bar` before we define `foo` then the compiler knows what `bar` is.
- This is enough for the compiler to know that the function call to `bar` in `foo` is correct.
- It is *highly recommended* to declare all your functions before defining them so that you avoid these types of problems.

Functions

What will happen? Why?

```
void foo(int) { cout << "int" << endl; }

void foo(double) { cout << "double" << endl; }

int main()
{
  foo(5);
  foo(2.7);
  foo(true);
}
```

Functions

Function pointers

```
int add(int x, int y)
{
  return x + y;
}

int sub(int x, int y)
{
  return x - y;
}
```

```
int main()
{
  // pointer to function taking
  // two int:s and returning int
  int (*ptr)(int, int);

  ptr = add;
  // print 2
  std::cout << ptr(1, 1) << std::endl;

  ptr = sub;
  // print 0
  std::cout << ptr(1, 1) << std::endl;
}
```

Functions

Function pointers

- In C++ there are two types of pointers: *data pointers* and *function pointers*
- Data pointers contain the address of some object (or collection of objects in the case of array pointers)

Functions

Function pointers

- But there are also *function pointers* in C++
- Function pointers contain the address of some *function* (machine code)
- A function pointer acts just as a function meaning we can call it and so on
- But instead of calling a fixed function it will call the one it points to
- A function pointer must specify the return type and parameters.

Functions

Function pointers

- Because of this, the syntax is quite complex:
  `int (*ptr)(int, int);`
- A function pointer must point to a function that have *exactly* the specified parameters and return type.
- We can also have pointers to functions that doesn't take parameters and doesn't return anything:
  `void (*ptr)();`
- We can also create *anonymous* function pointers:
  `void (*)()` and/or `void()`

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

```
struct My_Struct
{
  int a;
  int b;
  double c;
  char d;
};

My_Struct obj { 1, 2, 3.4, '5' };
```

Initialization

Aggregate initialization

```
struct My_Struct
{
  int a;
  int b;
  double c;
  char d;
};

My_Struct obj { 1, 2, 3.4, '5' };
```

Initialization

Aggregate initialization

```
struct My_Struct
{
  int a;
  int b;
  double c;
  char d;
};

My_Struct obj { 1, 2, 3.4, '5' };
```

Initialization

Aggregate initialization

```
struct My_Struct
{
  int a;
  int b;
  double c;
  char d;
};

My_Struct obj { 1, 2, 3.4, '5' };
```

Initialization

Aggregate initialization

```
struct My_Struct
{
  int a;
  int b;
  double c;
  char d;
};

My_Struct obj { 1, 2, 3.4, '5' };
```

Initialization

Be careful with paranthesis in initialization

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

```cpp
// function returning int
// taking no parameters
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?

```
int main()
{
  int x{};
  cout << x << " ";
  int y = 3.5;
  cout << y << " ";
  int z {3.5};
  cout << z << endl;
}
```

Value categories

What is the difference?

```
int x { 3 };

x = 7; // works
7 = x; // doesn't work

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

arr[2] = x + 12; // works
x + 12 = arr[2]; // doesn't work

x = int{}; // works
int{} = x; // doesn't work
```

Value categories

There seems to be two kinds of expressions here

Can be to the left

- `x`
- `arr[2]`
- `*ptr` (pointer)

Can *only* be to the right

- `7`
- `x + 12`
- `int{}`

Value categories

There seems to be two kinds of expressions here

Can be to the left

- `x`
- `arr[2]`
- `*ptr` (pointer)

left-hand-side **value** (lvalue)

Can *only* be to the right

- `7`
- `x + 12`
- `int{}`

right-hand-side **value** (rvalue)

Value categories

lvalues

- lvalues are expression that will refer to the same specific *object* every time we use it.
- So something that has a memory address and a name is always an lvalue, examples: `x`, `arr`, `std::cout`, etc.
- But things without a name can also be lvalues, for example: `arr[2]`, `*ptr` (dereference pointer) etc.
- We say that lvalues have *identity* (the expression refers to a specific object)

Value categories

rvalues

- rvalues are generally those values that are *not* lvalues.
- More specifically we can think of them as *temporary values*, meaning they have no identity (the expression refers to a specific *value* rather than object)
- For example: when evaluating the expression `x + 1` a new *temporary* value is created, so we never refer to the same *object* as earlier.
- Literals are rvalues: `5` `3.14f` `"my string"` etc.

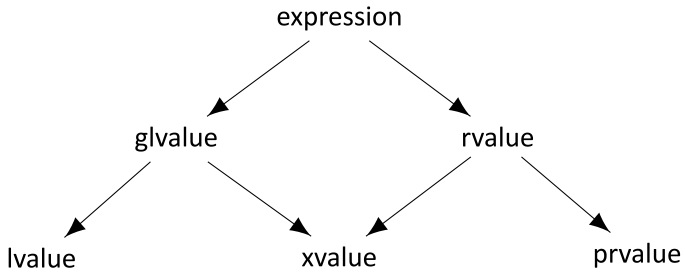
Value categories

Expressions

- each expression in C++ have a type: specifically the type that is returned from the expression
- example: `2*(1+1)` have the type `int`
- But they also have a *value category* which determines certain properties: can we assign to it? Does they value have identity?
- lvalues and rvalues are generally what is needed, but there are more fine-grained value categories as well.

Value categories

Expressions



Value categories

Expressions

- glvalue
- lvalue
- xvalue
- prvalue

Value categories

Expressions

- glvalue
  - **g**eneralied **l**eft-hand-size **value**;
  - denotes an object with identity
  - example: given a variable `x`, the expression `x` will be a glvalue
- lvalue
- xvalue
- prvalue

Value categories

Expressions

- glvalue
- lvalue
  - denote all glvalues that are not *xvalues*
- xvalue
- prvalue

Value categories

Expressions

- glvalue
- lvalue
- xvalue
  - exxpiring **value**
  - denotes an object bound to an rvalue reference (see next seminar for details)
  - example: `static_cast<int&&>(x)`, where `x` is of type `int`
- prvalue

Value categories

Expressions

- glvalue
- lvalue
- xvalue
- prvalue
  - **p**ure **r**ight-hand-side **value**
  - a value literal
  - the value of an expression
  - can be used to initialize glvalues
  - example: `5`, `true`, `nullptr`
  - example: `x+1`, where `x` is of type `int`

Value categories

Expressions

- glvalue
- lvalue
- xvalue
- prvalue

The term rvalue refers to both xvalues and prvalues.

LINKÖPING UNIVERSITY

Conversions

Promotions and narrowing conversions

Conversion rank:

```
bool < char < short < int < long < long long
```

← Narrowing (explicit) | Promotion (implicit) →

Conversions

Promotions and narrowing conversions

Conversion rank:

```
unsigned char < unsigned short < unsigned int < unsigned long < unsigned long long
```

← Narrowing (explicit) | Promotion (implicit) →

Conversions

Promotions and narrowing conversions

Conversion rank:

```
float < double < long double
```

← Narrowing (explicit) | Promotion (implicit) →

Conversions

Promotions and narrowing conversions

- There are many different numeric types in C++
- Mainly two categories: integers and floating-point numbers
- Within each category there are differently sized types that can be converted between each other
- The compiler is always allowed to implicitly *promote* a value to a larger type if it needs to
- But it is never allowed to silently perform a *narrowing* conversion (i.e. convert it to a smaller type)

Conversions

Promotions and narrowing conversions

- Promoting a type is always OK since whatever value we represent is guaranteed to fit in a larger type
- While narrowing conversions can be quite dangerous since certain values cannot be represented by a smaller type.
- For example: The larget value `char` can represent is `127`, so what will happen if I try to convert an `int` of value `378` to `char`?
- No clear answer since this is *undefined behaviour*.

Conversions

Implicit type conversions

- array-to-pointer and function-to-pointer
- promotions (integral and floating)
- integral and floating conversions
- boolean conversions

Conversions

Implicit type conversions

- array-to-pointer and function-to-pointer
  - lvalues of arrays or functions decays to pointers;
  - arrays becomes a pointer to the first element;
  - functions become pointers to the code.
- promotions (integral and floating)
- integral and floating conversions
- boolean conversions

Conversions

Implicit type conversions

- array-to-pointer and function-to-pointer
- promotions (integral and floating)
  - integral types smaller than `int` can be *promoted* into `int`;
  - `float` can be promoted to `double`;
  - enum types can be promoted to its underlying type.
- integral and floating conversions
- boolean conversions

Conversions

Implicit type conversions

- array-to-pointer and function-to-pointer
- promotions (integral and floating)
- integral and floating conversions
  - Coresponds to all non-promotions between integral or floating point types;
  - Conversion rank denotes the "size";
  - `long long` > `long` > `int` > `short` > `char` > `bool`.
  - `long double` > `double` > `float`
- boolean conversions

Conversions

Implicit type conversions

- array-to-pointer and function-to-pointer
- promotions (integral and floating)
- integral and floating conversions
- boolean conversions
  - integral types and pointers can be converted to `bool`;
  - all zero values (`0` and `nullptr`) are `false`;
  - all non-zero values are `true`.

Conversions

What will happen? Why?

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

Conversions

What will happen? Why?

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

Conversions

What will happen? Why?

```
void foo() { cout << "foo" << endl; }

int main()
{
  cout << foo << endl;
}
```

Conversions

What will happen? Why?

```
int main()
{
  int var (int());
  cout << var << endl;
}
```

Conversions

Most Vexing Parse

- This is sometimes called *the most vexing parse*;
- Declarations are prefered over definitions;
- Ambiguity is a problem in C++;
- A lot of ambiguity is resolved by using *brace-initialization* whenever possible.

LINKÖPING
UNIVERSITY

Memory Management & Pointers

What will happen? Why?

```
int& get()
{
  int x{5};
  return x;
}

int main()
{
  cout << get() << endl;
}
```

Memory Management & Pointers

What will happen? Why?

```
int const* get()
{
  return new int{5};
}

int main()
{
  cout << *get() << endl;
}
```

Memory Management & Pointers

Manual Memory Management

```
int const* get()
{
  return new int{5};
}

int main()
{
  int const* const x{get()};
  cout << x << endl;
  delete x;
}
```

Memory Management & Pointers

Pointers vs. Arrays

```
int main()
{
  int static_array[5];
  int* dynamic_array {new int[5]};
  cout << sizeof(static_array) << " ";
  cout << sizeof(dynamic_array) << endl;
  delete[] dynamic_array;
}
```

LINKÖPING UNIVERSITY

Command-Line Arguments

```
int main(int argc, char* argv[])
{
  if (argc != 3)
  {
    cerr << "Wrong argument count!" << endl;
    return 1;
  }
  for (int arg{}; arg < argc; ++arg)
    cout << argv[arg] << endl;
  return 0;
}
```

Command-Line Arguments

```
int main(int argc, char* argv[])
{
  if (argc != 3)
  {
    cerr << "Wrong argument count!" << endl;
    return 1;
  }
  for (int arg{}; arg < argc; ++arg)
    cout << argv[arg] << endl;
  return 0;
}
```

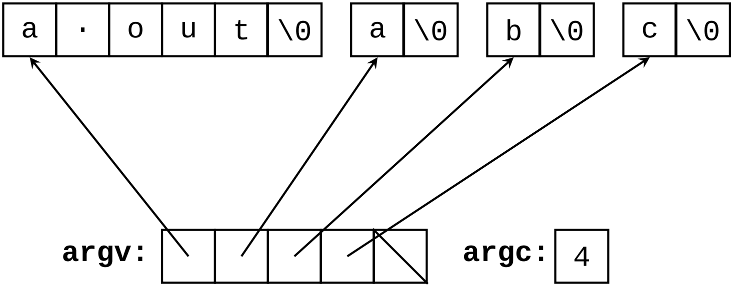
```
$ a.out a b c
```

Command-Line Arguments

```
$ a.out a b c
a.out
a
b
c
```

Command-Line Arguments

What is argv?



www.liu.se

LINKÖPING UNIVERSITY