TDDD38 - 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`.

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Data types

Type categories

There are four categories of types:

- Fundamental types
- Array types
- Class types
- Enum types

Data types

Type categories

There are four categories of types:

- Fundamental types
  - types that can be used directly;
  - basic building blocks of all other types;
  - commonly used for arithmetic operations;
  - examples: `int`, `double`, `char`, `bool`.
- Array types
- Class types
- Enum types

Data types

Type categories

There are four categories of types:

- Fundamental types
- Array types
  - represent arrays of a single type;
  - used for storing a fixed count of values;
  - there are better alternatives in modern C++;
  - example: `int array[3];`
- Class types
- Enum types

Data types

Type categories

There are four categories of types:

- Fundamental types
- Array types
- Class types
  - types composed of several different types;
  - can even contain functions;
  - all `class`, `struct` and `union` types.
- Enum types

Data types

Type categories

There are four categories of types:

- Fundamental types
- Array types
- Class types

```
struct Person
{
  string name; // class type
  int age; // fundamental type
  int get_age(){ return age; } // function
};
```

- Enum types

Data types

Type categories

There are four categories of types:

- Fundamental types
- Array types
- Class types

```
union JSON
{
    double val;
    char const* str;
    double get_value() { return val; }
};
```

- Enum types

Data types

Type categories

There are four categories of types:

- Fundamental types
- Array types
- Class types
- Enum types
  - a predefined set of discrete values;
  - each possible value has a name;
  - is an integral type;
  - two variations: unscoped and scoped.

Data types

Type categories

There are four categories of types:

- Fundamental types
- Array types
- Class types
- Enum types

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

Data types

Type categories

There are four categories of types:

- Fundamental types
- Array types
- Class types
- Enum types

```
enum class Status : char // scoped
{
  ERROR = -1,
  PENDING,
  GRANTED,
  DENIED
};
```

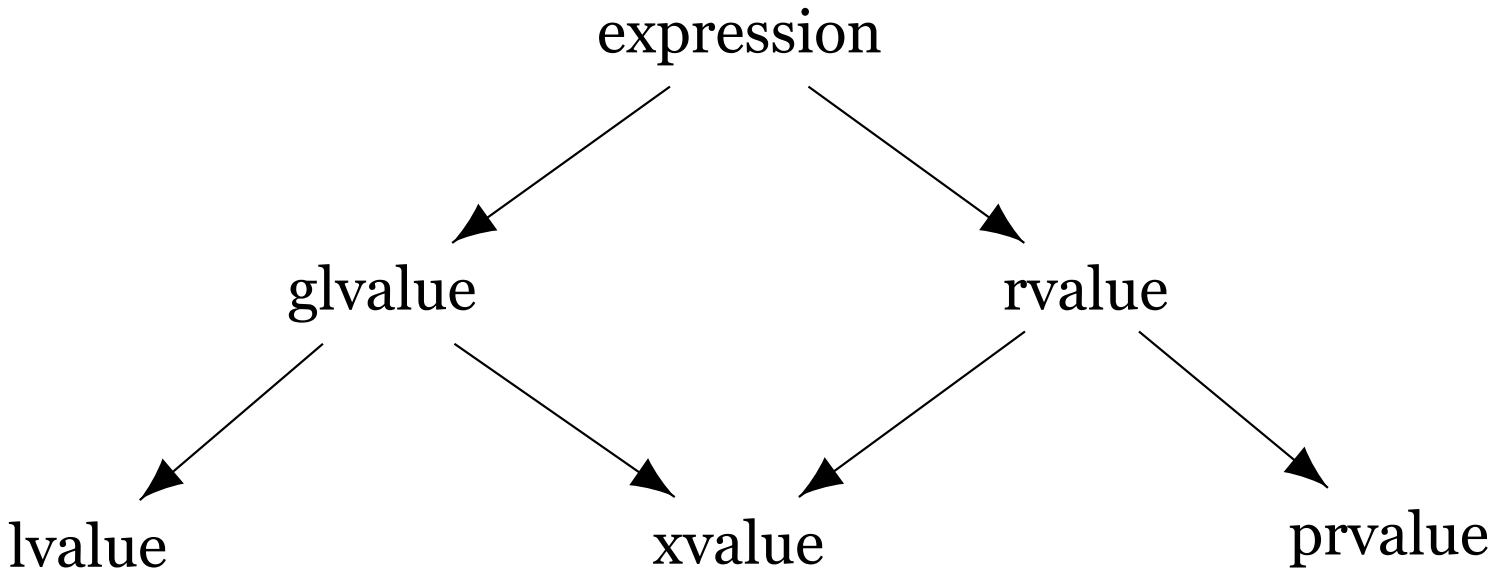
Data types

Value categories

- each expression in C++ have a type;
- the type of value that will be returned;
- example: `2*(1+1)` have the type `int`.

Data types

Value categories



Data types

Value categories

each expression is a part of exactly one value category;

- glvalue
- lvalue
- xvalue
- prvalue

Data types

Value categories

each expression is a part of exactly one value category;

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

Data types

Value categories

each expression is a part of exactly one value category;

- glvalue
- lvalue
  - **l**eft-hand-size **value**;
  - is a special case of glvalues;
  - denote all glvalues that are not *xvalues*;
  - the *name rule*: everything tha has a name is an lvalue.
- xvalue
- prvalue

Data types

Value categories

each expression is a part of exactly one value category;

- glvalue
- lvalue
- xvalue
  - e**x**piring **value**;
  - denote something temporary;
  - is a special case of glvalues;
  - an object created without a name;
  - example: `int{}`.
- prvalue

Data types

Value categories

each expression is a part of exactly one value category;

- 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`.

Data types

Value categories

each expression is a part of exactly one value category;

- glvalue
- lvalue
- xvalue
- prvalue

The term rvalue refers to both xvalues and prvalues.

Initialization

Ways of initialization

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

Initialization

Ways of initialization

- Copy initialization: `int x = 5;`
- Direct initialization: `int x(5);`
  - initialize an object by calling an appropriate constructor;
  - try to convert the supplied value to the same type as the object;
  - more permissive than *copy initialization*.
- Value initialization: `int x{};`
- List initialization: `int x{5};`

Initialization

Ways of initialization

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

Initialization

Ways of initialization

- Copy initialization: `int x = 5;`
- Direct initialization: `int x(5);`
- Value initialization: `int x{};`
- List initialization: `int x{5};`
  - will *copy initialize* if possible;
  - otherwise *value initialize*;
  - otherwise if the type is a class type it will try to initialize each member from the supplied arguments;
  - Narrowing conversions are prohibited during list initializations.

Initialization

Ways of initialization

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

It is highly recommended to use *brace-initialization* whenever possible.

Initialization

What will happen?

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

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
  - all non-promotions between integral or floating point types;
  - integral types uses integer conversion rank to choose;
  - `long long` > `long` > `int` > `short` > `char` > `bool`.
- 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;
}
```

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

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

What will happen? Why?

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

Functions

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.

1 Initial example
2 Data types
3 Initialization
4 Conversions
5 Functions
6 Memory Management & Pointers
7 Command-Line Arguments

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;
}
```

1 Initial example
2 Data types
3 Initialization
4 Conversions
5 Functions
6 Memory Management & Pointers
7 Command-Line Arguments

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;
}
```

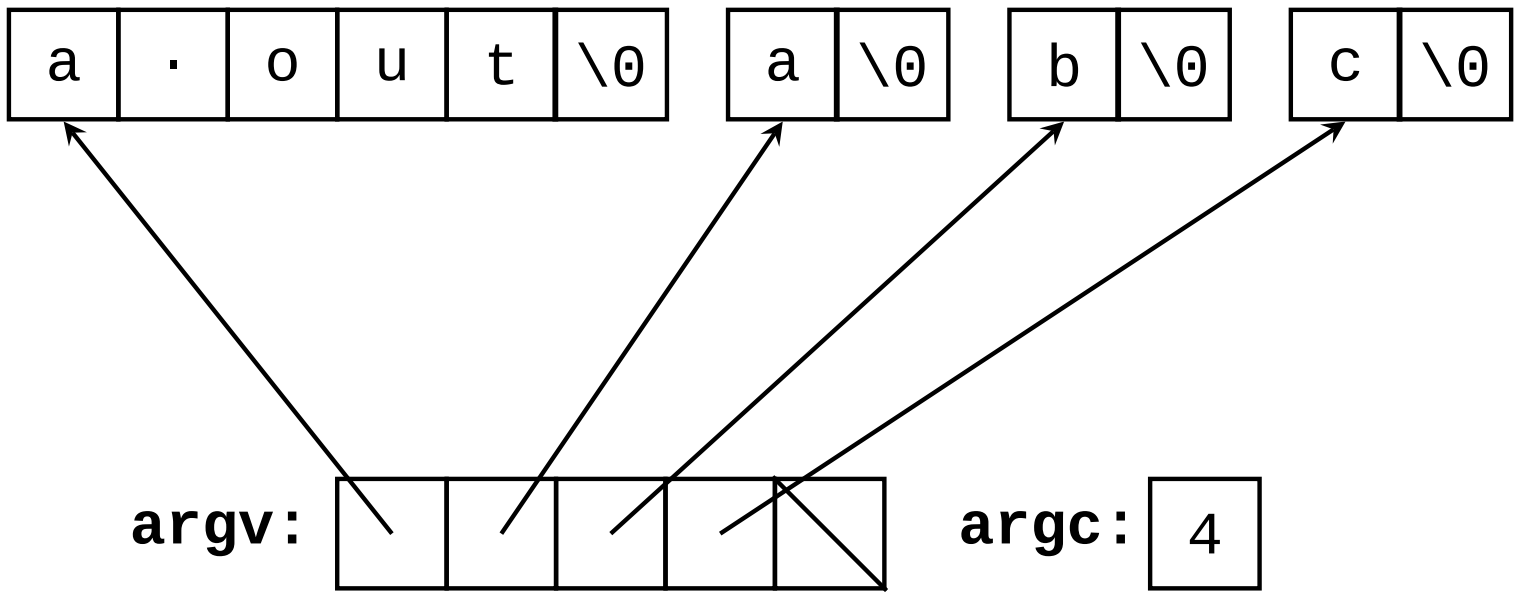
```
$ a.out a b c
```

Command-Line Arguments

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

Command-Line Arguments

What is argv?



www.liu.se

LINKÖPING UNIVERSITY