An introduction to C++

day 8

Sandro Andreotti, Chris Bielow

Bioinformatics Solution Center

Slides: Hannes Hauswedell

Timeframe

Day	Content	State
Monday	Recap week 1 & test	✓
Tuesday	Object-oriented programming 1, Essential operations, Regular types	✓
Wednesday	Memory management, lifetime, pointers	✓
Thursday	Object-oriented programming 2, move-semantics, forwarding	✓
Friday	Type conversions and casts + test	←

Conversion Introduction

Implicit Conversion

Explicit Conversion

Conversion Introduction

- **Reminder:** C++ is statically typed, i.e. the type of every object is known and fixed at compile-time. By default you cannot use type T1 in a place where type T2 is expected (if T1 != T2).
- Language is not very clear around all terms; in this lecture **anything that transforms an object of type T1 into an object of type T2 is a conversion**.
- There are *implicit conversions* defined on the next slide.
- All other forms of conversion are *explicit conversions*.

Conversion Introduction

- **Reminder:** C++ is statically typed, i.e. the type of every object is known and fixed at compile-time. By default you cannot use type T1 in a place where type T2 is expected (if T1 != T2).
- Language is not very clear around all terms; in this lecture **anything that transforms an object of type T1 into an object of type T2 is a conversion**.
- There are *implicit conversions* defined on the next slide.
- All other forms of conversion are *explicit conversions*.
- **Convertibility of `T1 → T2` is independent of convertibility `T2 → T1`!**

Conversion Introduction

Implicit Conversion

Explicit Conversion

Implicit Conversion - what?

Implicit conversions are performed whenever an expression of some type T1 is used in a context that does not accept that type, but accepts some other type T2;[...] An expression `e` is said to be implicitly convertible to `T2` if and only if `T2` can be copy-initialized from `e`, that is the declaration

```
T2 t = e;
```

is well-formed (can be compiled).

Multiple steps are allowed to perform an *implicit conversion sequence*, including *standard conversions* and *user-defined conversions* (the exact sequences are listed below).

from https://en.cppreference.com/w/cpp/language/implicit_conversion

Implicit Conversion - when?

Implicit conversions are performed whenever an expression of some type `T1` is used in context that does not accept that type, but accepts some other type `T2`; in particular:

- when passing a function argument, e.g. `void foo(T2); foo(T1{});`
- when passing an operand to an operator, e.g. `T2 & operator=(T2 const &)… T2 t; t = T1{};`
- when initializing a new object, including the return statement in a function, e.g. `T2 foo() { return T1{}; }`
- *when the expression is used in a switch statement (`T2` is integral type);*
- *when the expression is used in an if statement or a loop (`T2` is bool).*

from https://en.cppreference.com/w/cpp/language/implicit_conversion

Implicit Conversion - how

standard conversions include:

- numeric promotions/conversions: basically all arithmetic types and `bool` are implicitly convertible to each other; behaves *mostly as expected*, but information loss can occur (e.g. `size_t` → `bool`)
- pointer-conversions: e.g. ptr-to-derived → ptr-to-base-class (but not ←)
- array-to-pointer-conversion: `char[10]` → `char *`
- qualification conversions: e.g. adding `const`-ness

user-defined conversions:

- via a converting constructor in the target type; *or*
- via a user-defined conversion operator in the source type

Implicit Conversion - how

standard conversions, the good:

- `int &` ⟶ `int const &`
- `DerivedType *` ⟶ `BaseType *`

standard conversions, the bad:

- `bool` ⟶ `int`

standard conversions, the ugly ("narrowing"):

- `uint64_t` ⟶ `uint32_t`
- `float` ⟶ `int`

All compilers offer warning-flags that prevent some or all of these.

Implicit Conversion - how

```
class A
{
public:
    // Rule-of-six...

    size_t i{};
};

class B
{
public:
    // Rule-of-six...

    size_t j{};

    B(A const & a) { j = a.i; }
};
```

Converting constructors:

`B` provides a constructor that takes objects of type `A`:

```
A a;
// ...

B b1{a};
// or
B b2 = a;
```

```
void foo(B const & b) { /*…*/ }

foo(A{});
```

Implicit Conversion - how

```
class A
{
public:
    // Rule-of-six...
    size_t i{};

    operator B() const
    {
        B b; b.j = i; return b;
    }
};

class B
{
public:
    // Rule-of-six...
    size_t j{};
};
```

Conversion operator:

`A` provides an operator that converts to `B`.

```
A a;
// ...

B b1{a};
// or
B b2 = a;
```

```
void foo(B const & b) { /*…*/ }

foo(A{});
```

Conversion Introduction

Implicit Conversion

Explicit Conversion

Explicit Conversion - when?

1. When an object is *direct-initialised*:

```
class A { /*…*/ };
class B { /*…*/ };

A a{};

B b1{a};    // direct-init. considers implicit and explicit conversions
B b2(a);    // direct-init. (old-style)

B b3 = a;   // copy-init. only considers implicit conversions
```

2. When using explicit *casts*.

Explicit Conversion - what?

Explicit **user-defined conversions**:

- via a converting constructor in the target type; *or*
- via a user-defined conversion operator in the source type

Depending on cast:

- various other behaviours

Explicit Conversion - how

```
class A
{
public:
    // Rule-of-six...

    size_t i{};
};

class B
{
public:
    // Rule-of-six...

    size_t j{};

    explicit B(A const & a) {j = a.i;}
};
```

Converting constructors:

`B` provides an **explicit** constructor that takes objects of type `A`:

```
A a;
// ...

B b1{a};
// the following NOT
//B b2 = a;
```

```
void foo(B const & b) { /*…*/ }
A a;
foo(B{a}); //not: foo(a);
```

Explicit Conversion - how

```
class A
{
public:
    // Rule-of-six...
    size_t i{};

    explicit operator B() const
    {
        B b; b.j = i; return b;
    }
};

class B
{
public:
    // Rule-of-six...
    size_t j{};
};
```

Conversion operator:

`A` provides an **explicit** operator that converts to `B`.

```
A a;
// ...

B b1{a};
// the following NOT
//B b2 = a;
```

```
void foo(B const & b) { /*…*/ }

foo(B{A{}}); //not: foo(A{});
```

Conversion Introduction

Implicit Conversion

Explicit Conversion

Casts

`static_cast<TO>(FROM)`

Can do…

- implicit conversions[1]
- explicit user-defined conversions (via constructor or operator)
- down-casts of references/pointers (conversion from base-type to derived-type[2])
  → **no checks performed whether this is valid!**
- adding `&&` to type (in fact `std::move` == `static_cast<T&&>(T)`).
- conversion to/from strongly typed enums

```
foo(static_cast<B>(A{}));
int i = 10;
A* a = static_cast<A*>(&i); // Compile error (good!)
```

[1] Some implicit conversions are dangerous and produce warnings; performing them via `static_cast` silences the warning.
[2] The other way around than implicitly.

`dynamic_cast`

Can do…

- only works polymorphic types
- down-casts of references/pointers (conversion from base-type to derived-type[2])
  → **performs check whether this is valid!**

```
std::vector<ShapeInterface *> shapes;
// add Circles and Squares

for (ShapeInterface * p : shapes)
    if (dynamic_cast<Square*>(p) != nullptr)
        std::cout << "This is a square!\n";
```

Other casts

```
A a;
A const & ar = a;
```

`reinterpret_cast<B>(a)`:

- bit level reinterpretation of one type as another one
- no check, no actual conversions
- dangerous, but useful in certain high-performance use-cases

`const_cast<A &>(a2)`:

- can remove `const`-ness, but only when referring to something that isn't const in the first place.
- not used widely

C-Style cast:

- Syntax: `(B)a` or `B(a)`, e.g. `int a = (int)3.45f;` .
- attempts different possible C++ casts in this order (intent may be unclear!)

```
const_cast
static_cast
static_cast followed by const_cast
reinterpret_cast
reinterpret_cast followed by const_cast
```

- less restrictive than `static_cast`
- no compile time or run time check! ('segfaults' if you get it wrong...)

```
char c = 10;       // 1 byte
int *p = (int*)&c; // points to 4 bytes .. dereferencing this might fail
*p = 5; // stack corruption!
```

whereas

```
int *q = static_cast<int*>(&c); // compile-time error
```

Summary

- Use warning levels that prevent some of the dangerous implicit conversions.
- If you add user defined conversions (via constructor or operator) make them `explicit`.
- Use `static_cast` for most conversions.
- Use `dynamic_cast` for safe downcasts (when using dynamic polymorphism).