A note on Explicit Initialization in C++

Dirk Vermeir

July 12, 2003

Contents

Abstract

This note, which is also available in postscript (init-with-ass-stmt.ps) and pdf (init-with-ass-stmt.pdf) format, clarifies the difference between both forms of explicit initialization

```
T t = v; // assignment syntax
T t(v); // explicit syntax
```

1 Explicit Syntax

This is very simple: in a statement of the form

```
T t(v);
```

where `v` is of some type V, the object `t` is initialized by calling the constructor `T::T(V v)`.

See the first example .

2 Assignment Syntax

In the statement below, `t` is initialized using (the syntax of) an assignment statement.

```
T t = v;
```

For such a statement, the compiler will generate code that is equivalent with the following sequence of steps.

1. If necessary, i.e. if `v` is not of type T, convert `v` to a temporary T object `tmp_t`.
2. Initialize `t` using the copy constructor `T::T(const T&)` with *`tmp_t`* as argument.

In most cases, the compiler may optimize such that the effect of the assignment syntax is the same as that of the explicit syntax. This is the case, e.g. if there exists a (non-explicit) constructor `T::T(V)` (see the example), where we assume that `v` has type V.

3 Examples

Example 3.1 The usual case

The code is available in the file test1.C.

```
#include <iostream>
class T {
public:
  T(int i): data_(i) { std::cerr << "T::T(int)" << std::endl; }
  T(const T&) { std::cerr << "T::T(const T&)" << std::endl; }
private:
  int data_;
};

int
main(int, char**) {
T t1(3); // OK
T t2 = 3; // OK
}
```

The output of the above program is

```
T::T(int)
T::T(int)
```

indicating that, for the statement `T t2 = 3`, the compiler optimized away the theoretical (according to step 2) call to the copy constructor of T, by directly applying the convertor/constructor `T::T(int)` to `t2`.

Example 3.2 The assignment syntax

Even if, as in the above example , the compiler does not generate a call to the copy constructor, it must be available. This is illustrated in the following program (available as test-2.C) which does not compile.

```
#include <iostream>
class T {
public:
  T(int i): data_(i) { std::cerr << "T::T(int)" << std::endl; }
private:
  T(const T&) { std::cerr << "T::T(const T&)" << std::endl; }
  int data_;
};

int
main(int, char**) {
T t1(3); // OK
T t2 = 3; // Error: T::T(const T&) private.
}
```

For the above program , the compiler issues the following error message

```
test-2.C: In function ‘int main(int, char**)’:
test-2.C:6: ‘T::T(const T&)’ is private
test-2.C:13: within this context
test-2.C:13:   initializing temporary from result of ‘T::T(int)’
```

4 When the distinction is important

In the code below, available as test-3.C, the initialization of `s2` yields a compilation error.

```
#include <string>
#include <iostream>

class T {
public:
  explicit T(int i): data_(i) { std::cerr << "T::T(int)" << std::endl; }
  template <typename X>
  operator X() const {
    X x;
    std::cerr << "T::operator X()" << std::endl;
    return x;
    }
private:
  T(const T&) { std::cerr << "T::T(const T&)" << std::endl; }
  int data_;
};

int
main(int, char**) {
T t(3); // OK
std::string s1 = t; // OK.
std::string s2(t); // Error: ambiguity.
}
```

The reason is that std::string has two constructors with a single parameter, taking resp. a const char* or std::string argument.

```
std::string::string(const char*);
std::string::string(const std::string&*);
```

Due to the member template, the compiler can convert to both const char* and std::string, with similar "matching quality". Hence the ambiguity.

Removing the offending line, leaves the code below.

```
#include <string>
#include <iostream>

class T {
public:
  explicit T(int i): data_(i) { std::cerr << "T::T(int)" << std::endl; }
  template <typename X>
  operator X() const {
    X x;
    std::cerr << "T::operator X()" << std::endl;
    return x;
    }
private:
  T(const T&) { std::cerr << "T::T(const T&)" << std::endl; }
  int data_;
};

int
main(int, char**) {
T t(3);
std::string s = t;
}
```

which compiles ok and produces the output

```
T::T(int)
T::operator X()
```

showing once more that the compiler optimized away the theoretical call to T's copy constructor, in this case by directly putting the result of `T::operator std::string(t)` in `s`.