LEARN C++
Skill up with our free tutorials

13.14 — Converting constructors, explicit, and delete

ALEX JULY 25, 2021

By default, C++ will treat any constructor as an implicit conversion operator. Consider the following case:

```cpp
#include <cassert>
#include <iostream>

class Fraction
{
private:
 int m_numerator;
 int m_denominator;

public:
 // Default constructor
 Fraction(int numerator = 0, int denominator = 1)
    : m_numerator(numerator), m_denominator(denominator)
 {
  assert(denominator != 0);
 }

 // Copy constructor
 Fraction(const Fraction& copy)
  : m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
 {
  // no need to check for a denominator of 0 here since copy must already be a valid Fraction
  std::cout << "Copy constructor called\n"; // just to prove it works
 }

 friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
 int getNumerator() { return m_numerator; }
 void setNumerator(int numerator) { m_numerator = numerator; }
};

void printFraction(const Fraction& f)
{
 std::cout << f;
}

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
 out << f1.m_numerator << "/" << f1.m_denominator;
 return out;
}

int main()
{
 printFraction(6);

 return 0;
}
```

Although function printFraction() is expecting a Fraction, we've given it the integer literal 6 instead. Because Fraction has a constructor willing to take a single integer, the compiler will implicitly convert the literal 6 into a Fraction object. It does this by initializing printFraction() parameter f using the Fraction(int, int) constructor.

Consequently, the above program prints:

```
6/1
```

This implicit conversion works for all kinds of initialization (direct, uniform, and copy).

Constructors eligible to be used for implicit conversions are calledconverting constructors (or conversion constructors).

The explicit keyword

While doing implicit conversions makes sense in the Fraction case, in other cases, this may be undesirable, or lead to unexpected behaviors:

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

class MyString
{
private:
 std::string m_string;
public:
 MyString(int x) // allocate string of size x
 {
  m_string.resize(x);
 }

 MyString(const char* string) // allocate string to hold string
value
 {
  m_string = string;
 }

 friend std::ostream& operator<<(std::ostream& out, const MyString&
s);

};

std::ostream& operator<<(std::ostream& out, const MyString& s)
{
 out << s.m_string;
 return out;
}

void printString(const MyString& s)
{
 std::cout << s;
}

int main()
{
 MyString mine = 'x'; // Will compile and use MyString(int)
 std::cout << mine << '\n';

 printString('x'); // Will compile and use MyString(int)
 return 0;
}
```

In the above example, the user is trying to initialize a string with a char. Because chars are part of the integer family, the compiler will use the converting constructor MyString(int) constructor to implicitly convert the char to a MyString. The program will then print this MyString, to unexpected results. Similarly, a call to printString('x') causes an implicit conversion that results in the same issue.

One way to address this issue is to make constructors (and conversion functions) explicit via the explicit keyword, which is placed in front of the function's name. Constructors and conversion functions made explicit will not be used for *implicit* conversions or copy initialization:

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

class MyString
{
private:
 std::string m_string;
public:
 // explicit keyword makes this constructor ineligible for implicit conversions
 explicit MyString(int x) // allocate string of size x
 {
  m_string.resize(x);
 }

 MyString(const char* string) // allocate string to hold string value
 {
  m_string = string;
 }

 friend std::ostream& operator<<(std::ostream& out, const MyString& s);

};

std::ostream& operator<<(std::ostream& out, const MyString& s)
{
 out << s.m_string;
 return out;
}

void printString(const MyString& s)
{
 std::cout << s;
}

int main()
{
 MyString mine = 'x'; // compile error, since MyString(int) is now explicit and nothing will match
this
 std::cout << mine;

 printString('x'); // compile error, since MyString(int) can't be used for implicit conversions

 return 0;
}
```

The above program will not compile, since MyString(int) was made explicit, and an appropriate converting constructor could not be found to implicitly convert 'x' to a MyString.

However, note that making a constructor explicit only prevents *implicit* conversions. Explicit conversions (via casting) are still allowed:

```
std::cout << static_cast<MyString>(5); // Allowed: explicit cast of 5 to
MyString(int)
```

Direct or uniform initialization will also still convert parameters to match (uniform initialization will not do narrowing conversions, but it will happily do other types of conversions).

```
MyString str{'x'}; // Allowed: initialization parameters may still be implicitly converted to
match
```

Best practice

Consider making your constructors and user-defined conversion member functions explicit to prevent implicit conversion errors.

The delete keyword

In our MyString case, we really want to completely disallow 'x' from being converted to a MyString (whether implicit or explicit, since the results aren't going to be intuitive). One way to partially do this is to add a MyString(char) constructor, and make it private:

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

class MyString
{
private:
 std::string m_string;

 MyString(char) // objects of type MyString(char) can't be constructed from outside the class
 {
 }

public:
 // explicit keyword makes this constructor ineligible for implicit conversions
 explicit MyString(int x) // allocate string of size x
 {
  m_string.resize(x);
 }

 MyString(const char* string) // allocate string to hold string value
 {
  m_string = string;
 }

 friend std::ostream& operator<<(std::ostream& out, const MyString& s);

};

std::ostream& operator<<(std::ostream& out, const MyString& s)
{
 out << s.m_string;
 return out;
}

int main()
{
 MyString mine('x'); // compile error, since MyString(char) is private
 std::cout << mine;
 return 0;
}
```

However, this constructor can still be used from inside the class (private access only prevents non-members from calling this function).

A better way to resolve the issue is to use the "delete" keyword to delete the function:

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

class MyString
{
private:
 std::string m_string;

public:
 MyString(char) = delete; // any use of this constructor is an error

 // explicit keyword makes this constructor ineligible for implicit conversions
 explicit MyString(int x) // allocate string of size x /
 {
  m_string.resize(x);
 }

 MyString(const char* string) // allocate string to hold string value
 {
  m_string = string;
 }

 friend std::ostream& operator<<(std::ostream& out, const MyString& s);

};

std::ostream& operator<<(std::ostream& out, const MyString& s)
{
 out << s.m_string;
 return out;
}

int main()
{
 MyString mine('x'); // compile error, since MyString(char) is deleted
 std::cout << mine;
 return 0;
}
```

When a function has been deleted, any use of that function is considered a compile error.

Note that the copy constructor and overloaded operators may also be deleted in order to prevent those functions from being used.

Next lesson
13.15 Overloading the assignment operator

Back to table of contents

Previous lesson
13.13 Copy initialization

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ co

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest