User-Defined Conversions

The material in this handout is collected from the following references:

- Sections 7.5.4 and Section 14.9.1 of the text book C++ Primer.
- Various sections of Effective C++.
- Various sections of More Effective C++.

Introduction

C++ allows compilers to perform *implicit promotions and conversions* between built-in numerical types. Because of its C roots, C++ allows silent promotions such as allowing a `char` value to be promoted to an `int` value and silent conversions such as allowing a `short` value to be converted to a `double` value. This is why we can pass a `char` to a function that takes a `double` and still have the call succeed. Similarly, when we create our own types, we can provide member functions that compilers can use for implicit type conversions. Two kinds of functions allow compilers to perform such conversions: ***single argument conversion constructors*** and ***implicit type conversion operators***.

Single argument conversion constructors

A ***single argument conversion constructor*** is a constructor that can called with only one argument. Such a constructor may declare only a single parameter or it may declare multiple parameters, with each parameter after the first having a default value. In the following definition of class `Stopwatch`:

```
1  class Stopwatch {
2  public:
3    Stopwatch(int);
4    // other functions declared that are not relevant to discussion ...
5  private:
6    int seconds;
7  };
8
9  Stopwatch::Stopwatch(int s) : seconds(s) {
10   // function body is intentionally empty ...
11 }
```

the constructor

```
1  Stopwatch::Stopwatch(int);
```

is an example of a *single argument conversion constructor*. The presence of the conversion constructor allows the following code to compile by implicitly converting an `int` value to a `Stopwatch` object:

```
void foo(Stopwatch const& rsw) {
  // some stuff here that uses rsw ...
}

int main() {
  int x {10};

  // constructs a temporary Stopwatch object with data member seconds
  // equal to x then, the newly generated temporary Stopwatch object is
  // passed to foo
  foo(x);
}
```

The compiler uses conversion constructor `Stopwatch::Stopwatch(int)` to convert `int` argument `x` in function call `foo(x)` to an unnamed `Stopwatch` object on the stack. Next, the compiler will initialize parameter `rsw` in the definition of function `foo` as an alias to this unnamed `Stopwatch` object.

Consider the following two classes: class `Str` encapsulates a `std::string` object while class `Student` encapsulates a `Str` object:

```
// dummy class to encapsulate std::string type
class Str {
public:
  Str(std::string const& s); // conversion ctor from std::string to Str

  // other member functions which are not of relevant here ...
private:
  std::string a_str;
};

Str::Str(std::string const& s) : a_str(s) {
  // empty ...
}

// encapsulate information about Student ...
class Student {
public:
  Student(Str const& s); // conversion ctor from Str to Student

  // other member functions which are not of relevant here ...
private:
  Str name; // student's have names
  // other data members not relevant here ...
};

Student::Student(Str const& s) : name(s) {
  // empty ...
}
```

Here, we initialize a `Student` object using a `std::string` object:

```
std::string john{"John"};
Student s(john); // ok: std::string object is automatically converted
                 // to a temporary Str object
```

The initialization of object `s` is perfectly legal; the compiler automatically creates a temporary `Str` object from the given `std::string` object called `john` which is then used to construct object `s` of type `Student`.

Only one class-type conversion is allowed

The compiler will implicitly apply only one class-type conversion. The following code is in error because it requires two implicit conversions:

```
// error: requires two user-defined conversions:
// 1st conversion: convert C-style string to std::string object
// 2nd conversion: convert std::string object to Str object
Student s1("John");
```

If we wanted to construct a `Student` object using a C-style string, we can do so by explicitly converting the C-style string to either a `std::string` or a `Str` object:

```
// ok:
// first, C-style string "John" is explicitly converted to unnamed
// temporary std::string object.
// next, this temporary std::string object is implicitly converted to
// unnamed temporary Str object.
// finally, temporary Str object is used to invoke conversion ctor
// Student::Student(Str const&)
Student s1(std::string("John"));

// ok:
// first, C-style string "John" is implicitly converted to unnamed
// temporary std::string object.
// next, this temporary std::string object is explicitly converted to
// temporary Str object via the ctor Str::Str(std::string const&)
// then, this temporary Str object is used to invoke conversion
// ctor Student::Student(Str const&)
Student s2(Str("John"));
```

Implicit type conversion operator

An implicit type conversion operator is a member function that has a weird signature: identifier `operator` followed by the type specification. We're not allowed to specify a type for the return value, because the return value's type is just the name of the function. For example, to allow `Stopwatch` objects to be implicitly converted to `int`s, the implicit type conversion operator is declared in class `Stopwatch` like this:

```
class Stopwatch {
public:
  Stopwatch(int); // conversion constructor
  operator int () const; // implicit type conversion operator

  // some other functions declared that are not relevant to the discussion
...
private:
  int seconds;
};
```

Outside the class definition, the definition of the implicit type conversion operator would look like this:

```
Stopwatch::operator int () const {
  return secs;
}
```

This function would be implicitly invoked in situations where the compiler expects an `int` but the compiler sees an operand of type `Stopwatch`:

```
void boo(int x) {
  // do some stuff with parameter x ...
}

int main() {
  Stopwatch sw(30);

  // Compiler will use Stopwatch::operator int () to convert Stopwatch
  // expression in function call to an int value which is then used to
  // initialize the parameter x of function boo
  boo(sw);

  // Compiler will implicitly convert left operand sw to an int, add the
  // two operands and use the result to initialize the int object y
  int y = sw + 40;
}
```

Be wary of conversion functions

Implicit conversions in C that result in loss of data such as the conversion of `int`s, `short`s, or `double`s to `char` are also present in C++. There's nothing we can do about such promotions and conversions because they're hard coded into the language. When we add our own types, however, we have more control, because we can choose whether to provide the functions compilers are allowed to use for implicit type conversions. We'll now provide rationales for why, as a general rule of thumb, it is preferable to not implement conversion functions in classes.

Replace implicit type conversion operators with equivalent functions

As a general rule of thumb, class designers must be wary of implicit type conversion operators because they end up being called when clients neither want nor expect them to be. The result can be incorrect and unintuitive program behavior that is difficult to debug. For example, consider the scenario where the client would like to print `Stopwatch` objects as if they were a built-in type, as in:

```
Stopwatch sw(30);
std::cout << sw; // should print "30"
```

Further suppose that the class designer hasn't implemented `operator<<` overload for `Stopwatch` objects. The client would probably expect the attempt to print `sw` to fail, because there is no appropriate `operator<<` to call. However, the C++ compiler, faced with a call to function called `operator<<` that takes a `Stopwatch`, would find that no such function exists, but being hardworking, it then would try to find an acceptable sequence of implicit type conversions it

could apply to make the call succeed. The compiler will discover that it could make the call succeed by implicitly converting `sw` to an `int` by calling `Stopwatch::operator int()`. This example code fragment demonstrates the disadvantage of implicit type conversions: their presence can lead to the wrong function being called [i.e., one other than the intended one].

The solution is to replace the implicit type conversion operator with equivalent functions that don't have the syntactically magic names. For example, to allow conversion of a `Stopwatch` to an `int`, replace the implicit type conversion function with a member function called something like `to_int`:

```
class Stopwatch {
public:
  Stopwatch(int);         // conversion constructor
  operator int () const; // implicit type conversion operator
  int to_int() const;    // converts Stopwatch to int

  // some other functions declared that are not relevant to the discussion
...
private:
  int secs;
};
```

`Stopwatch::to_int` being a member function must be called explicitly:

```
Stopwatch sw(30);
std::cout << sw; // error! no operator<< defined for Stopwatch
std::cout << sw.to_int(); // fine, prints sw as an int
```

In most cases, the inconvenience of having to call conversion functions explicitly is more than compensated for by the fact that unintended functions can no longer be silently invoked. In general, the more experience C++ programmers have, the more likely they are to eschew implicit type conversion operators. Members of the standard C++ library committee are amongst the most experienced and perhaps that's why the `std::string` class contains no implicit type conversion from a `string` object to a `char*`. Instead, there's an explicit member function, `std::string::c_str` that performs the conversion.

Judiciously declare conversion constructors as `explicit`

Conversion constructors are useful and practical in many scenarios. C++ standard library type `std::string` provides a constructor `string::string(const char* s)` that converts a null-terminated C-style string pointed to by `s` into a `std::string` object, as in:

```
#include <string>

std::string name("John");
```

In other cases, a conversion constructor may not be useful - code involving class-type conversions using conversion constructors is unlikely to behave in a satisfactory manner and may cause significant run-time inefficiencies. Consider a sample class definition of a container representing `int` arrays that can change in size:

```
// example from Item 5 of More Effective C++

```

```
3  #include <cstddef> // for size_t
4
5  class Array {
6  public:
7    // construct array using a range of array indices
8    Array(int const* low_bound, int const* high_bound);
9    Array(size_t sz); // conversion ctor
10   // other ctors and dtor here ...
11
12   // overloaded subscript operator ...
13   int& operator[](size_t idx);
14   int const& operator[](size_t idx) const;
15
16   size_t size() const; // size of array
17
18   // other member functions not relevant here ...
19
20 private:
21   size_t len;
22   int *ptr;
23 };
```

The first constructor in the class definition allows clients to construct an `Array` object from a range of indices of another `int` array. Because the constructor requires two arguments, it doesn't qualify for use as a type conversion function. The second constructor can be used as a type conversion function because it allows clients to define `Array` objects by specifying only the number of elements in the array. However, this constructor can cause client code to behave in unexpected ways.

For example, consider a client-defined function to compare two `Array` objects:

```
1  // client defined non-member function to compare two Array objects
2  bool operator==(Array const& lhs, Array const& rhs) {
3    if (lhs.size() != rhs.size()) {
4      return false;
5    }
6
7    size_t equiv_cnt {0};
8    for (size_t i {0}; i < lhs.size(); ++i) {
9      if (lhs == rhs[i]) { // oops!!! "lhs" should be "lhs[i]"
10       ++equi_cnt;
11     }
12   }
13
14   if (equiv_cnt == lhs.size()) { // all elements in lhs and rhs
15     return true;                 // have equivalent values
16   }
17   return false;
18 }
```

We intended to compare each element of `lhs` to the corresponding element in `rhs`, but on line 9 we accidentally omitted the subscripting syntax when we typed in `lhs` instead of `lhs[i]`. We'd expect this mistyped comparison expression to elicit all sorts of errors; however, the compiler will not complain at all. That's because it sees a call to `operator==` with arguments of type `Array` [for `lhs`] and `int` [for `rhs`]. Though there is no `operator==` function taking these two types,

the compiler notices it can convert the `int` into an `Array` object by calling the conversion constructor `Array::Array(size_t)`. Each iteration through the loop thus compares the contents of `lhs` with the contents of a temporary array of size `rhs[i]` [whose contents are presumably uninitialized]. The code will thus generate recursive calls to non-member function `operator==` producing code that doesn't do the correct thing.

We can avoid the problems prevalent in the previous code fragment by using keyword `explicit`. Constructors can be declared `explicit`, and if they are, compilers are prohibited from invoking them for purposes of implicit type conversions. Explicit conversions are still legal, however:

```
class Array {
public:
  // conversion ctor - notice use of keyword "explicit"
  explicit Array(size_t sz);
  // other member functions not relevant here ...
private:
  // data members here ...
};

// ok: explicit ctors can be used as usual for object construction
Array a(10), b(10);

// error!!!: no way to implicitly convert int to Array
if (a == b[i]) {
  // some code here ...
}

// ok: conversion from int to Array is explicit
// but the logic of code is suspect ...
if (a == Array(b[i])) {
  // some code here ...
}

// ok: conversion from int to Array is explicit
// but the logic of code is suspect ...
if (a == static_cast<Array>(b[i])) {
  // some code here ...
}

// ok: explicit conversion from int to Array using C-style casts
// logic of code is still suspect ...
if (a == (Array)b[i]) {
  // some code here ...
}
```

How to decide when to make conversion constructors `explicit` or non-`explicit`

The final word: Be judicious on deciding whether a conversion constructor in a user-defined class must be `explicit` or non-`explicit`. If there is a conceptual relationship between the argument to a conversion constructor and the class type, clients might benefit from the implicit conversion. On the other hand, if there is no conceptual relationship between the argument to a conversion constructor and the class type, clients can write more intuitive and correct code using explicit conversions as when required.

The C++ standard library container `std::vector` models dynamic arrays. To avoid the problems described earlier, library designers prohibit the implicit conversion of `int`s to `vector`s by ensuring `std::vector` declares an `explicit` conversion constructor `std::vector<T>::vector(size_t n)` which creates a vector with `n` elements, each element of type `T` created by `T`'s default constructor. On the other hand, the close conceptual relationship between C-style strings which have type `char const*` and the C++ standard library type `std::string` has led library designers to decide on a non-explicit conversion constructor `std::string::string(char const* s)`.