Type Conversion

CIS-3012, C++ Programming

Vermont State University

Peter Chapin

C-Style Type Conversion

- In C, use this syntax to explicitly request the conversion of one type to another:

```
long x = 42L;    // ‘L’ suffix means “long.”
int y = (int)x;  // A type “cast.”
```

- The target type in parenthesis goes in front of what is to be converted.
- The C-style cast operator is available in C++ for C compatibility…
  - … but C++ has other, better ways to express type conversions.

The Problem with Type Casts

- C-Style type casts are a free-for-all:
  - Can be used to do *safe* casts that are well-defined and well-behaved
  - Can be used to do *unsafe* casts that are extremely system-dependent or undefined
  - Can be used to *cast away const* (i.e., remove constant-ness from objects and pointers)
- This cast is well-defined:

```
double pi = 3.14159;
int approximate_pi = (int)pi;  // Converts to 3
```

Strange C-Style Casts

- What do these do?

```
double pi = 3.14159;
int approximate_pi = *(int *)&pi;

unsigned long hardware_address = 0xFFFF001C;
unsigned char *device_register = (unsigned char *)hardware_address;
*device_register |= 0x02;
```

- The second example is a normal thing to do in C programs.
- The first example is just broken on machines with 32-bit integers (why?)

C++ Type Conversion Operators

- `static_cast<T>( x )`
  - Converts x to type T safely if possible; compile error otherwise.
- `const_cast<T>( x )`
  - Can only be used to add/remove constant-ness. Any other conversion is an error.
- `reinterpret_cast<T>( x )`
  - “Reinterprets” x as a T. Used for dangerous conversions.
- `dynamic_cast<T>( x )`
  - Used to downcast in an inheritance hierarchy.

Static Cast

- Static casts can only be used for well-defined conversions.

```
long x = 42L;
int  y = static_cast<int>( x );  // No error or warning.

double pi = 3.14159;
int approximate_pi = static_cast<int>( pi );  // No error or warning.

double pi = 3.14159;
int *p = static_cast<int *>( &pi );
  // Error! Can’t statically cast incompatible pointer types.
```

Const Cast

- Can only be used to remove (or add) constant-ness.

```
void f( const char *pc ) {
    char *p = const_cast<char *>( pc );  // No error or warning.
    *p = ‘x’;   // Trying to modify a constant!
}
```

- There are times when temporarily removing constant-ness is useful…
  - … when you know what the pointer is really pointing at, and you are going after a special effect.
- That doesn’t mean you want an unsafe, ill-defined conversion of the data!

Reinterpret Cast

- Reinterpret casts can reinterpret the bits of one value as if they were another type.
- Can be dangerous. Usually very non-portable. Sometimes necessary.
- Often used with pointer types:

```
unsigned char *hardware_register =
  reinterpret_cast<unsigned char *>( 0xFFFF007E );
```

- Can be used for special effects:

```
double pi = 3.14159;         // Double precision is (almost always) 64 bits.
unsigned long raw_bits =     // Assume unsigned long is 64 bits.
  reinterpret_cast<unsigned long>( pi );
```

Dynamic Cast

- We won't discuss dynamic casts in this class!
  - They are used to *downcast* in an inheritance hierarchy.

Implicit Conversions

- The C++ compiler will convert between certain types implicitly
  - There is a school of thought that says all implicit conversions are bad for you
  - … but many are convenient and completely safe
- Integer promotions
- Arithmetic conversions
- Signed/Unsigned conversions
- User-defined conversions

Integer Promotions

- Integer promotions are implicit conversions from a narrow integer to a wider integer. They are completely safe.
  - **short** → **int**
  - **int** → **long**
  - etc…

```
int count_special_things( const std::vector<Thing> &vec, const Thing &a_thing );

long count = count_special_things( some_vector, some_thing );
```

Implicit conversion from **int** to **long**.
No possible loss of information on any platform.
No compiler warning.

Arithmetic Conversions

- Includes Integer promotions, but also *unsafe* conversions
  - **long → int**
  - **int → short**
  - **unsigned int → int**
  - etc…

```
int count_special_things( const std::vector<Thing> &vec, const Thing &a_thing );

short count = count_special_things( some_vector, some_thing );
```

Implicit conversion from **int** to **short**.
Possible loss of information on some platforms.
Compiler warning possible (likely on platforms that actually experience loss).

Conversions in Expressions

- *When types are mixed, the narrow type is promoted to the wider type*
  - The full rules are more complicated, but the statement above is the idea

```
int x = 42;
long y = 84;
double z = 3.14159;

x = x + y;
  // x is promoted to long, the addition is done as long.
  // The resulting long is converted (with possible loss) back to int for the assignment.

z = x + y;
  // x is promoted to long, the addition is done as long.
  // The resulting long is converted to double for the assignment.
```

Signed/Unsigned Conversions

- Consider int and unsigned int…
  - The standard requires they have the same number of bits
  - BUT… they have overlapping ranges
  - <u>Conversion in either direction is unsafe</u>
  - *Avoid mixing signed and unsigned types*!
  - Fix any compiler warnings that arise from doing so
    - Many bugs arise from such mixing

The type alias `size_t`

- C (and C++) define a type alias in various headers: `size_t`
  - It is an <u>unsigned integral type</u> suitable for measuring the size of objects in memory
  - On 64-bit systems it is usually `unsigned long`. On 16-bit systems it is usually `unsigned int`. *You should always use `size_t` where it is appropriate* (for portability, don't try to use the underlying type directly)

```
#include <cstring>  // C++'s version of C's header <string.h>

int length = std::strlen( s );
```

Warning: Mixing signed and unsigned types!
Warning: Possible loss of precision (64-bit `size_t` vs 32-bit `int`)
<u>*Fix warnings like these!*</u>

Why?

- Why does C allow unsafe implicit conversions?
  - C++ does it for C compatibility… except when the uniform initialization syntax is used… C compatibility isn't an issue in that situation.
- Good question!
  - Probably a "mistake" in the design of C.
  - Many, many bugs and security problems arise because of this. The language should never have allowed it.
  - Modern C (and C++) compilers <u>produce warnings</u> for most unsafe conversions… if you ask for them.
  - *Always use* `–Wall` *when compiling with g++!* <u>*Treat warnings as errors!*</u>

User-Defined Conversions

- If you don’t like implicit conversions, you will hate this…
- C++ allows you to define implicit conversions for your classes
- There are two directions:
  - Implicitly converting something to your class type
  - Implicitly converting an instance of your class to something else
- The two directions are handled a little differently

Constructors w/ One Parameter

- Unless marked as **`explicit`**, *constructors that can be called with one argument are taken as an implicit conversion from the type of the argument to the type of the class*.

```
std::string s = “Hello, World”;
```

String literals have type: `const char *`

There is a constructor for `std::string` that takes a `const char *` parameter.

Compiler uses that constructor to create a `std::string` temporary…
… and then initializes `s` from that temporary.

In this case, the temporary can be (and most likely is) optimized away.

Another (more typical?) Example

- This shows the implicit conversion being used with an argument

```
void process_string( const std::string &process_me );

process_string( “Hello, World” );
```

Here the compiler uses the single parameter constructor taking `const char *` to create a `std::string` temporary for `process_string`.

Notice this only works because the parameter of `process_string` is a reference to `const`. Otherwise, the compiler won’t bind that reference to a temporary.

Converting From Class

- To implicitly convert a class instance to some other type, use conversion operators:

```
class BigFloat {
public:
    // …
    operator double( ) const;
};
```

```
BigFloat::operator double( ) const
{
    double result;

    // Do what must be done

    return result;
}
```

Now We Can Do…

- The class BigFloat is an infinite precision floating point type.

```
BigFloat pi{ “3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510” };
  // I assume BigFloat has a constructor that can handle the string above.

double approximate_pi = pi;   // Calls BigFloat::operator double( )
                              // No warning about precision loss
                              // Compiler doesn’t know what the conversion does


void process_number( double process_me );

process_number( pi );         // Calls BigFloat::operator double( )
                              // Again, no warning about precision loss
```