Type Conversion | Coursera

Type Conversion

Type Conversion

The next natural question is “what happens if you have a binary operator, and its operands have different types?” For example, if a is an int and b is a double, then what type is a + b? The answer to this question depends on the types involved.

Fundamentally, the first thing that must happen is that the two operands must be converted to the same type. Most operations can only be performed on operands of the same type. The processor has one instruction to add two 32-bit integers, a different instruction to add two 16-bit integers, a third one to add two 32-bit floats, a fourth to add two 64-bit doubles, and so on. The compiler must translate your code into one of these instructions, so, it must pick one of them and arrange to have the inputs in a proper format in order to be able to perform the math.

When the two operands have different types, the compiler attempts to add a *type conversion* (sometimes called a *type promotion*) to make the types the same. If no type conversion is possible, the compiler will issue an error message and refuse to compile the code. When the compiler inserts a type conversion, it typically must add instructions to the program which cause the processor to explicitly change the bit representation from the size and representation used by the original type to the size and representation used by the new type. The compiler chooses which operand to convert based on what gives the “best” answer. In our int + double example, the compiler will convert the int to a double to avoid losing the fractional part of the number.

There are four common ways that the bit representations must be changed to convert from one type to another during a type promotion. When converting from a smaller signed integer type to a longer signed integer, the number must be sign extended—the sign bit (most significant bit) must be copied an appropriate number of times to fill in the additional bits. When converting from a smaller unsigned integer type to a longer unsigned integer type, the number must be zero extended—the additional bits are filled in with all zeros. The third common way that the bit representation can be changed during an automatic conversion happens when a longer integer type is converted to a shorter integer type. Here, the bit pattern is truncated to fit—the most significant bits are thrown away, and only the least significant bits are retained.

The fourth way that the bit representation may need to be changed is to fully calculate what the representation of the value is in the new type. For example, when converting from an integer type to a real type, the compiler must insert an instruction which requests that the CPU compute the floating point (binary scientific notation) representation of that integer.

There are other cases where a type conversion does not need to alter the bit pattern, instead just changing how it is interpreted. For example, converting from a **signed int** to an **unsigned int** leaves the bit pattern unchanged. However, if the value was originally negative, it will now be interpreted as a large positive number. Consider the following code:

```
1
2
3
4
5
6
7
8
unsigned int bigNum = 100;
int littleNum = -100;
if (bigNum > littleNum) {
  printf("Obviously, 100 is bigger than -100!\n");
}
else {
  printf("Something unexpected has happened!\n");
}
```

When this code runs, it prints “Something unexpected has happened!”. The bit pattern of littleNum (which has a leading 1 because it is negative) is preserved; the value is changed to a number larger than 100 (because under an unsigned interpretation, this leading 1 indicates a very large number). We will note that the compiler produces a warning (an indication that you probably did something bad—which means you should go fix your code!) for this behavior, as comparing signed integers to unsigned integers is typically a bad idea for exactly this reason.

When you declare a variable and assign it a particular type, you specify how you would like the data associated with that variable—the bit pattern “in the box”—to be interpreted for the entirety of its life span. There are some occasions, however, when you or the

compiler may have to temporarily treat the variable as though it were of another type. When a programmer does this, it is called *casting* and when a compiler does it, it is called *type conversion* or *type promotion*. It is extremely important to understand when to do the former and when the compiler is doing the latter because it can often be the cause of confusion and consequently errors. We will note that while understanding when to cast is important, understanding that you should generally not cast is even more important—sprinkling casts into your code to make errors go away indicates that you are not actually fixing your code, but rather hiding the problems with it.