some operands to different types so that the hardware will be able to evaluate the expression. If we add a 16-bit `short` and a 32-bit `int`, for example, the compiler will arrange for the `short` value to be converted to 32 bits. If we add an `int` and a `float`, the compiler will arrange for the `int` to be converted to `float` format. This conversion is a little more complicated, since `int` and `float` values are stored in different ways.

Because the compiler handles these conversions automatically, without the programmer's involvement, they're known as ***implicit conversions***. C also allows the programmer to perform ***explicit conversions***, using the cast operator. I'll discuss implicit conversions first, postponing explicit conversions until later in the section. Unfortunately, the rules for performing implicit conversions are somewhat complex, primarily because C has so many different arithmetic types.

Implicit conversions are performed in the following situations:

- When the operands in an arithmetic or logical expression don't have the same type. (C performs what are known as the ***usual arithmetic conversions***.)
- When the type of the expression on the right side of an assignment doesn't match the type of the variable on the left side.
- When the type of an argument in a function call doesn't match the type of the corresponding parameter.
- When the type of the expression in a `return` statement doesn't match the function's return type.

We'll discuss the first two cases now and save the others for Chapter 9.

The Usual Arithmetic Conversions

The usual arithmetic conversions are applied to the operands of most binary operators, including the arithmetic, relational, and equality operators. For example, let's say that `f` has type `float` and `i` has type `int`. The usual arithmetic conversions will be applied to the operands in the expression `f + i`, because their types aren't the same. Clearly it's safer to convert `i` to type `float` (matching `f`'s type) rather than convert `f` to type `int` (matching `i`'s type). An integer can always be converted to `float`; the worst that can happen is a minor loss of precision. Converting a floating-point number to `int`, on the other hand, would cost us the fractional part of the number. Worse still, we'd get a completely meaningless result if the original number were larger than the largest possible integer or smaller than the smallest integer.

The strategy behind the usual arithmetic conversions is to convert operands to the "narrowest" type that will safely accommodate both values. (Roughly speaking, one type is narrower than another if it requires fewer bytes to store.) The types of the operands can often be made to match by converting the operand of the narrower type to the type of the other operand (this act is known as ***promotion***). Among the most common promotions are the ***integral promotions***, which convert a character or short integer to type `int` (or to `unsigned int` in some cases). Q&A