reinterpret_cast from one pointer type to another or from one reference type to another is valid, so long as the cast does not drop a cv-qualifier (which would be ill formed). It is only an *access* through the pointer or reference that might be invalid, leading to undefined behavior. The general rule is that an access to an object through the result of a **reinterpret_cast**<T*> is valid if and only if an object of type T exists at that address at the time it is accessed. Most of the pitfalls described below are violations of this concise, general, and widely applicable rule.

1. **Using reinterpret_cast for object conversions** — A **reinterpret_cast** operates between pointer types, between reference types, between pointer-to-member types, and between pointer types and integral types, but not between other object types. It is ill formed to use **reinterpret_cast**s to perform type conversions, even among types for which conversions exist. We cannot, for example, **reinterpret_cast** an **int** to a **float** or vice versa, nor can we **reinterpret_cast** a **prvalue** such as 3.14 to a reference of any kind:

```
struct Class1 { explicit Class1(int); };  // explicitly convertible from int

float         rc1 = reinterpret_cast<float>(3);             // Error
int           rc2 = reinterpret_cast<int>(3.0);             // Error
const double& rc3 = reinterpret_cast<const double&>(3.14);  // Error
int&&         rc4 = reinterpret_cast<int&&>(3.14);          // Error, prvalue
int           rc5 = reinterpret_cast<int>(3);               // OK, no-op
unsigned      rc6 = reinterpret_cast<unsigned>(3);          // Error
Class1        rc7 = reinterpret_cast<Class1>(5);            // Error

float         sc1 = static_cast<float>(3);             // OK, but unnecessary
int           sc2 = static_cast<int>(3.0);             // OK,  "       "
const double& sc3 = static_cast<const double&>(3.14);  // OK,  "       "
int&&         sc4 = static_cast<int&&>(3.14);          // OK, temporary obj
int           sc5 = static_cast<int>(3);               // OK, no-op
unsigned      sc6 = static_cast<unsigned>(3);          // OK, but unnecessary
Class1        sc7 = static_cast<Class1>(5);            // OK
```

   Note that all of the ill-formed uses of **reinterpret_cast** above are valid uses of **static_cast**.

2. **Accessing objects of unrelated types via reinterpret_cast** — Although **reinterpret_cast** between incompatible pointer and reference types is always valid, undefined behavior can arise when attempting to dereference such a converted pointer or reference: Unless there is somehow a valid object of the appropriate type at that address, accessing a value stored there has undefined behavior.

   To illustrate the austerity of this rule, consider that even though two different trivial standard-layout types, e.g., A and B below, might have precisely the same layout