`alignof` User-Defined Types

When applied to user-defined types, alignment is always at least that of the strictest alignment of any of its arguments’ base or member objects. Compilers will by default avoid nonessential padding because any extra padding would be wasteful of memory, e.g., cache:

```
struct S0 { };                         // sizeof(S0) is  1; alignof(S0) is  1
struct S1 { char c; };                 // sizeof(S1) is  1; alignof(S1) is  1
struct S2 { short s; };                // sizeof(S2) is  2; alignof(S2) is  2
struct S3 { char c; short s; };        // sizeof(S3) is  4; alignof(S3) is  2
struct S4 { short s1; short s2; };     // sizeof(S4) is  4; alignof(S4) is  2
struct S5 { int i; char c; };          // sizeof(S5) is  8; alignof(S5) is  4
struct S6 { char c1; int i; char c2; }; // sizeof(S6) is 12; alignof(S6) is  4
struct S7 { char c; short s; int i; }; // sizeof(S7) is  8; alignof(S7) is  4
struct S8 { double d; };               // sizeof(S8) is  8; alignof(S8) is  8
struct S9 { double d; char c; };       // sizeof(S9) is 16; alignof(S9) is  8
struct SA { long double ld; };         // sizeof(SA) is 16; alignof(SA) is 16
struct SB { long double ld; char c; }; // sizeof(SB) is 32; alignof(SB) is 16
```

~~The sizes of~~ empty types, such as `S0` in the example above, ~~are~~ defined to have the size and alignment of 1 to ensure that each object and member subobject of type `S0` has a unique address. However, if an empty type is used as a base, the size of the derived type will not be affected (with some exceptions) due to the **empty-base optimization**:

```
struct D0 : S0 { int i; };  // sizeof(D0) is 4; alignof(D0) is 4
```

The alignment of the base type always ~~affects the derived type’s alignment. However, this effect is observable for an empty base only if it is~~ ~~**over-aligned**~~; see Section 2.1.“**alignas**” on page 168:

```
struct alignas(8) E { };   // sizeof(E)  is 8; alignof(E)  is 8
struct D1 : E { int i; };  // sizeof(D1) is 8; alignof(D1) is 8
```

Compilers are permitted to increase alignment — e.g., in the presence of virtual functions, which typically implies a virtual function table pointer — but have certain restrictions on padding. For example, they must ensure that each comprised type is itself sufficiently aligned. Furthermore, sufficient padding must be added so that the alignment of the parent type divides its size, ensuring that storing multiple instances in an array does not require any padding between array elements, which is explicitly prohibited by the Standard. In other words, the following identities hold for all types, `T`, and positive integers, `N`:

```
#include <cstddef>  // std::size_t

template <typename T, std::size_t N>
void f()
{
    static_assert(0 == sizeof(T) % alignof(T), "guaranteed");

    T a[N];
    static_assert(N == sizeof(a) / sizeof(*a), "guaranteed");
}
```