Chapter 1

Safe Features

ch-safe
sec-safe-cpp11 Intro text should be here.

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional
sec-conditional-cpp11 Intro text should be here.

The (Compile-Time) `alignof` Operator

alignof

The keyword `alignof` serves as a compile-time operator used to query the **alignment requirements** of a type on the current platform.

Description

description

The `alignof` operator, when applied to a type, evaluates to an **integral constant expression** that represents the **alignment requirements** of its argument type. Similar to `sizeof`, the (compile-time) value of `alignof` is of type `std::size_t`; unlike `sizeof` (which can accept an arbitrary expressions), `alignof` is defined (in the C++ Standard) on only a type identifier but often works on expressions anyway (see *Annoyances* on page 12). The argument type, `T`, supplied to `alignof` must be either a **complete type**, a **reference type**, or an **array type**. If `T` is a **complete type**, the result is the alignment requirement for `T`. If `T` is a **reference type**, the result is the alignment requirement for the referenced type. If `T` is an **array type**, the result is the alignment requirement for every element in the array[1]:

```
static_assert(alignof(short)    == 2, "");  // complete type   (sizeof is 2)
static_assert(alignof(short&)   == 2, "");  // reference type  (sizeof is 2)
static_assert(alignof(short[5]) == 2, "");  // array type      (sizeof is 2)
static_assert(alignof(short[])  == 2, "");  // array type      (sizeof fails)
```

`alignof` Fundamental Types

nof-fundamental-types

Like their size, the alignment requirements of a `char`, `signed char`, and `unsigned char` are all guaranteed to be 1 (i.e., 1-byte aligned) on every conforming platform. For any other fundamental or pointer type `FPT`, `alignof(FPT)` (like `sizeof(FPT)`) is platform-dependent but is typically approximated well by the type's **natural alignment** — i.e., `sizeof(FPT) == alignof(FPT)`:

```
static_assert(alignof(char)   == 1, "");  // guaranteed to be 1
static_assert(alignof(short)  == 2, "");  // platform-dependent
static_assert(alignof(int)    == 4, "");  //    "          "
static_assert(alignof(double) == 8, "");  //    "          "
static_assert(alignof(void*)  >= 4, "");  //    "          "
```

`alignof` User-Defined Types

of-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. Empty types are defined to have a size (and alignment) of 1 to ensure that every object has a unique address.[2] Compilers

[1] According to the C++11 Standard, "An object of **array type** contains a contiguously allocated non-empty set of `N` subobjects of type `T`" (**cpp11**, section 8.3.4, "Arrays," paragraph 1, p. 188). Note that, for every type `T`, `sizeof(T)` is always a multiple of `alignof(T)`; otherwise, storing multiple `T` instances in an array would be impossible without padding, and the Standard explicitly prohibits padding between array elements.

[2] An exception is made for an object of a type derived from an empty (base) class in that neither the size nor the alignment of the derived object is affected by the derivation:

will (by default) avoid nonessential padding because any extra padding would be wasteful of (e.g., cache) memory[3]:

```
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; };           // sizeof(SA) is 16; alignof(SA) is 16
struct SB { long double; char c};     // sizeof(SB) is 32; alignof(SB) is 16
```

use-cases

Use Cases

type-during-development

Probing the alignment of a type during development

Both `sizeof` and `alignof` are often used informally during development and debugging to confirm the compiler's understanding of those attributes for a given type on the current platform. For example:

```
#include <iostream>  // std::cout
```

```
struct S { int i; }      // size = 4; alignment = 4
struct E { };            // size = 1; alignment = 1
struct D : E { int i }; // size = 4; alignment = 4
```

[3]Compilers are permitted to increase alignment (e.g., in the presence of virtual functions) but have certain restrictions on padding. For example, they must ensure that each comprised type is itself sufficiently aligned and that the alignment of the parent type divides its size. This ensures that the fundamental identity for arrays holds for all types, `T`, and positive integers, `N`:

```
T a[N]; static_assert(n == sizeof(a) / sizeof(*a));  // guaranteed
```

The alignment of user-defined types can be made artificially stricter (but not weaker) using the `alignas` (see "**??**" on page **??**) specifier. Also note that, for **standard-layout types**, the address of the first member object is guaranteed to be the same as that of the parent object:

```
struct S { int i; }
class T { public: S s; }
T t;
static_assert(&t.s == &t,     "");  // guaranteed
static_assert(&t.s == &t.s.i, "");  // guaranteed
```

This property also holds for (e.g., anonymous) unions:

```
struct { union { char c; float f; double d; } } u;
static_assert(&u == &u.c, "");  // guaranteed
static_assert(&u == &u.f, "");  // guaranteed
static_assert(&u == &u.d, "");  // guaranteed
```

```
void f()
{
    std::cout << " sizeof(double): " <<  sizeof(double) << '\n';  //  always 8
    std::cout << "alignof(double): " << alignof(double) << '\n';  // usually 8
}
```

Printing the size and alignment of a `struct` along with those of each of its individual data members can lead to the discovery of suboptimal ordering of data members (resulting in wasteful extra padding). As an example, consider two `struct`s, `Wasteful` and `Optimal`, having the same three data members but in different order:

```
struct Wasteful
{
    char   d_c;  // size =  1;  alignment = 1
    double d_d;  // size =  8;  alignment = 8
    int    d_i;  // size =  4;  alignment = 4
};              // size = 24;  alignment = 8

struct Optimal
{
    double d_d;  // size =  8;  alignment = 8
    int    d_i;  // size =  4;  alignment = 4
    char   d_c;  // size =  1;  alignment = 1
};              // size = 16;  alignment = 8
```

Both `alignof(Wasteful)` and `alignof(Optimal)` are `8` but `sizeof(Wasteful)` is `24`, whereas `sizeof(Optimal)` is only `16`. Even though these two `struct`s contain the very same data members, the individual alignment requirements of these members forces the compiler to insert more total padding between the data members in `Wasteful` than is necessary in `Optimal`:

```
struct Wasteful
{
    char   d_c;           // size =  1;  alignment = 1
    char   padding_0[7];  // size =  7
    double d_d;           // size =  8;  alignment = 8
    int    d_i;           // size =  4;  alignment = 4
    char   padding_1[4];  // size =  4
};                        // size = 24;  alignment = 8

struct Optimal
{
    double d_d;           // size =  8;  alignment = 8
    int    d_i;           // size =  4;  alignment = 4
    char   d_c;           // size =  1;  alignment = 1
    char   padding_0[3];  // size =  3
};                        // size = 16;  alignment = 8
```

is-sufficiently-aligned

Determining if a given buffer is sufficiently aligned

The `alignof` operator can be used to determine if a given (e.g., `char`) buffer is suitably aligned for storing an object of arbitrary type. As an example, consider the task of creating a **value-semantic** class, `MyAny`, that represents an object of arbitrary type[4]:

```
void f()
{
    MyAny obj = 10;               // can be initialized with values of any type
    assert(obj.as<int>() == 10);  // inner data can be retrieved at runtime

    obj = std::string{"hello"};   // can be reassigned from a value of any type
    assert(obj.as<std::string>() == "hello");
}
```

A straightforward implementation of `MyAny` would be to allocate an appropriately sized block of dynamic memory each time a value of a new type is assigned. Such a naive implementation would force memory allocations even though the vast majority of values assigned in practice are small (e.g., fundamental types), most of which would fit within the space that would otherwise be occupied by just the pointer needed to refer to dynamic memory. As a practical optimization, we might instead consider reserving a small buffer (say, roughly[5] 32 bytes) within the footprint of the `MyAny` object to hold the value provided (1) it will fit and (2) the buffer is sufficiently aligned. The natural implementation of this type — the union of a char array and a `struct` (containing a `char` pointer and a size) — will naturally result in the minimal alignment requirement of the `char*` (i.e., 4 on a 32-bit platform and 8 on a 64-bit one)[6]:

```
class MyAny  // nontemplate class
{
    union
    {
```

[4]The C++17 Standard Library provides the (nontemplate) class `std::any`, which is a type-safe container for single values of *any* **regular type**. The implementation strategies surrounding alignment for `std::any` in both `libstdc++` and `libc++` closely mirror those used to implement the simplified `MyAny` class presented here. Note that `std::any` also records the current `typeid` (on construction or assignment) so that it can implement a `const` template member function, `bool is<T>() const`, to query, at runtime, whether a specified type is currently the active one:

```
void f(const std::any& object)
{
    if (object.is<int>()) { /* ... */ }
}
```

[5]We would likely choose a slightly larger value, e.g., 35 or 39, if that space would otherwise be filled with essential padding due to overall alignment requirements.

[6]We could, in addition, use the `alignas` attribute to ensure that the minimal alignment of `d_buffer` was at least 8 (or even 16):

```
// ...
alignas(8) char d_buffer[39];  // small buffer aligned to (at least) 8
// ...
```

```
        struct
        {
            char*       d_buf_p;  // pointer to dynamic memory if needed
            std::size_t d_size;   // for d_buf_p; same alignment as (char*)
        } d_imp;  // Size/alignment of d_imp is sizeof(d_buf_p) (e.g., 4 or 8).

        char d_buffer[39];         // small buffer aligned as a (char*)
    };  // Size of union is 39; alignment of union is alignof(char*).

    bool d_onHeapFlag;             // boolean (discriminator) for union (above)

public:
    template <typename T>
    MyAny(const T& x);               // (member template) constructor

    template <typename T>
    MyAny& operator=(const T& rhs);  // (member template) assignment operator

    template <typename T>
    const T& as() const;             // (member template) accessor

    // ...

};  // Size of MyAny is 40; alignment of MyAny is alignof(char*) (e.g., 8).
```

The (templated) constructor[7] of `MyAny` can then decide (potentially at compile time) whether to store the given object `x` in the internal small buffer storage or on the heap, depending on `x`'s size and alignment:

```
template <typename T>
MyAny::MyAny(const T& x)
{
    if (sizeof(x) <= 39 && alignof(T) <= alignof(char*))
    {
        // Store x in place in the small buffer.
        new(d_buffer) T(x);
        d_onHeapFlag = false;
    }
    else
    {
        // Store x on the heap and a pointer to it in the small buffer.
        d_imp.d_buf_p = new T(x);
        d_imp.d_size = sizeof(x);
        d_onHeapFlag = true;
    }
}
```

[7]In a real-world implementation, a *forwarding reference* would be used as the parameter type of `MyAny`'s constructor to *perfectly forward* the argument object into the appropriate storage; see "**??**" on page **??**.

Using the (compile-time) `alignof` operator in the constructor above to check whether the alignment of `T` is compatible with the alignment of the small buffer is necessary to avoid attempting to store overly aligned objects in place — even if they would fit in the 39-byte buffer. As an example, consider `long double`, which on typical platforms has both a size and alignment of `16`. Even though `sizeof(long double)` (16) is not greater than 39, `alignof(long double)` (16) is greater than that of `d_buffer` (8); hence, attempting to store an instance of `long double` in the small buffer, `d_buffer`, might — depending on where the `MyAny` object resides in memory — result in **undefined behavior**. User-defined types that either contain a `long double` or have had their alignments artificially extended beyond 8 bytes are also unsuitable candidates for the internal buffer even if they might otherwise fit:

```
struct Unsuitable1 { long double d_value };
    // Size is 16 (<= 39), but alignment is 16 (> 8).

struct alignas(32) Unsuitable2 { };
    // Size is  1 (<= 39), but alignment is 32 (> 8).
```

Monotonic memory allocation

tonic-memory-allocation

A common pattern in software — e.g., request/response in client/server architectures — is to quickly build up a complex data structure, use it, and then quickly destroy it. A **monotonic allocator** is a special-purpose memory allocator that returns a monotonically increasing sequence of addresses into an arbitrary buffer, subject to specific size and alignment requirements.[8] Especially when the memory is allocated by a single thread, there are prodigious[9] performance benefits to having unsynchronized raw memory be taken directly off the (always hot) program stack. In what follows, we will provide the building blocks of a monotonic memory allocator wherein the `alignof` operator plays an essential role.

As a practically useful example, suppose that we want to create a lightweight `MonotonicBuffer` class template that will allow us to allocate raw memory directly from the footprint of the object. Just by creating an object of an (appropriately sized) instance of this type on the program stack, memory will naturally come from the stack. For didactic reasons, we will start with a first pass at this class — ignoring alignment — and then go back and fix it using `alignof` so that it returns properly aligned memory:

```
template <std::size_t N>
struct MonotonicBuffer  // first pass at a monotonic memory buffer
{
    char  d_buffer[N];  // fixed-size buffer
    char* d_top_p;      //  next available address

    MonotonicBuffer() : d_top_p(d_buffer) { }
```

[8] C++17 introduces an alternate interface to supply memory allocators via an abstract base class. The C++17 Standard Library provides a complete version of standard containers using this more interoperable design in a sub-namespace, `std::pmr`, where `pmr` stands for **polymorphic memory resource**. Also adopted as part of C++17 are two concrete memory resources, `std::pmr::monotonic_buffer_resource` and `std::pmr::unsynchronized_pool_resource`.

[9] see **lakos16**

```
        // Initialize the next available address to be the start of the buffer.

    template <typename T>
    void* allocate()              // BAD IDEA --- doesn't address alignment
    {
        void* result = d_top_p;   // Remember the current next-available address.
        d_top_p += sizeof(T);     // Reserve just enough space for this type.
        return result;            // Return the address of the reserved space.
    }
};
```

MonotonicBuffer is a class template with one integral template parameter that controls the size of the d_buffer member from which it will dispense memory. Note that, while d_buffer has an alignment of 1, the d_top_p member, used to keep track of the next available address, has an alignment that is typically 4 or 8 (corresponding to 32-bit and 64-bit architectures, respectively). The constructor merely initializes the next-address pointer, d_top_p, to the start of the local memory pool, d_buffer[N]. The interesting part is how the allocate function manages to return a sequence of addresses corresponding to objects allocated sequentially from the local pool:

```
MonotonicBuffer<20> mb;  // On a 64-bit platform, the alignment will be 8.
char*   cp = static_cast<char*  >(mb.allocate<char  >());  // &d_buffer[ 0]
double* dp = static_cast<double*>(mb.allocate<double>());  // &d_buffer[ 1]
short*  sp = static_cast<short* >(mb.allocate<short >());  // &d_buffer[ 9]
int*    ip = static_cast<int*   >(mb.allocate<int   >());  // &d_buffer[11]
float*  fp = static_cast<float* >(mb.allocate<float >());  // &d_buffer[15]
```

The predominant problem with this first attempt at an implementation of allocate is that the addresses returned do not necessarily satisfy the minimum alignment requirements of the supplied type. A secondary concern is that there is no internal check to see if sufficient room remains. To patch this faulty implementation, we will need a function that, given an initial address and an alignment requirement, returns the amount by which the address must be rounded up (i.e., necessary padding) for an object having that alignment requirement to be properly aligned:

```
std::size_t calculatePadding(const char* address, std::size_t alignment)
    // Requires: alignment is a (non-negative, integral) power of 2.
{
    // rounding up X to N (where N is a power of 2): (x + N - 1) & ~(N - 1)
    const std::size_t maxA = alignof(std::max_align_t);
    const std::size_t a = reinterpret_cast<std::size_t>(address) & (maxA - 1);
    const std::size_t am1 = alignment - 1;
    const std::size_t alignedAddress = (a + am1) & ~am1;  // round up
    return alignedAddress - a;                            // return padding
}
```

Armed with the calculatePadding helper function (above), we are all set to write the final (correct) version of the allocate method of the MonotonicBuffer class template:

```
template <typename T>
void* MonotonicBuffer::allocate()
```

```
{
    // Calculate just the padding space needed for alignment.
    const std::size_t padding = calculatePadding(d_top_p, alignof(T));

    // Calculate the total amount of space needed.
    const std::size_t delta = padding + sizeof(T);

    // Check to make sure the properly aligned object will fit.
    if (delta > d_buffer + N - d_top_p)  // if (Needed > Total - Used)
    {
        return 0;  // not enough properly aligned unused space remaining
    }

    // Reserve needed space; return the address for a properly aligned object.
    void* alignedAddress = d_top_p + padding;  // Align properly for T object.
    d_top_p += delta;                          // Reserve memory for T object.
    return alignedAddress;                     // Return memory for T object.
}
```

Using this corrected implementation that uses `alignof` to pass the alignment of the supplied type `T` to the `calculatePadding` function, the addresses returned from the benchmark example (above) would be different[10]:

```
MonotonicBuffer<20> mb;  // Assume 64-bit platform (8-byte aligned).
char*   cp = static_cast<char*   >(mb.allocate<char   >());  // &d_buffer[ 0]
double* dp = static_cast<double*>(mb.allocate<double>());  // &d_buffer[ 8]
short*  sp = static_cast<short* >(mb.allocate<short >());  // &d_buffer[16]
int*    ip = static_cast<int*   >(mb.allocate<int    >());  // 0 (out of space)
bool*   bp = static_cast<bool*   >(mb.allocate<bool   >());  // &d_buffer[18]
```

In practice, an object that allocates memory, such as a `vector` or a `list`, will be constructed with an object that allocates and deallocates memory that is guaranteed to be either **maximally aligned**, **naturally aligned**, or sufficiently aligned to satisfy an optionally specified alignment requirement.

Finally, instead of returning a null pointer when the buffer was exhausted, we would typically have the concrete allocator fall back to a geometrically growing sequence of dynamically allocated blocks; the `allocate` method would then fail (i.e., a `std::bad_alloc` exception would somehow be thrown) only if all available memory were exhausted and the **`new` handler** were unable to acquire more memory yet still opted to return control to its caller.

[10]Note that on a 32-bit architecture, the `d_top_p` character pointer would be only four-byte aligned, which means that the entire buffer might be only four-byte aligned. In that case, the respective offsets for `cp`, `dp`, `sp`, `ip`, and `bp` in the example for the aligned use case might sometimes instead be 0, 4, 12, 16, and `nullptr`, respectively. If desired, we can use the `alignas` attribute/keyword to artificially constrain the `d_buffer` data member always to reside on a maximally aligned address boundary, thereby improving consistency of behavior, especially on 32-bit platforms.

Annoyances

annoyances-alignof

alignof (unlike sizeof) is defined only on types

defined-only-on-types

The (compile-time) `sizeof` operator comes in two different forms: one accepting a *type* and the other accepting an *expression*. The C++ Standard currently requires that `alignof` support only the former[11]:

```
static_assert(sizeof(int)  == 4, "");     // OK, int is a type.
static_assert(alignof(int) == 4, "");     // OK, int is a type.
static_assert(sizeof(3 + 2))  == 4, "");  // OK, 3 + 2 is an expression.
static_assert(alignof(3 + 2)) == 4, "");  // Error, 3 + 2 is not a type.
```

This asymmetry can result in a need to leverage `decltype` (see "**??**" on page **??**) when inspecting an expression instead of a type:

```
int f()
{
    enum { e_SUCCESS, e_FAILURES } result;
    std::cout << "size: " << sizeof(result) << '\n';
    std::cout << "alignment:" << alignof(decltype(result)) << '\n';
}
```

The same sort of issue occurs in conjunction with modern **type inference** features such as `auto` (see "**??**" on page **??**) and generic lambdas (see "**??**" on page **??**). As a real-world example, consider the generic lambda (C++14) being used to introduce a small *local function* that prints out information regarding the size and alignment of a given `object`, likely for debugging purposes:

```
auto printTypeInformation = [](auto object)
{
    std::cout << "     size: " << sizeof(object) << '\n'
              << "alignment: " << alignof(decltype(object)) << '\n';
};
```

Because there is no explicit type available within the body of the `printTypeInformation` lambda,[12] a programmer wishing to remain entirely within the C++ Standard[13] is forced to use the `decltype` construct explicitly to first obtain the type of `object` before passing it on to `alignof`.

---

[11] Although the Standard does not require `alignof` to work on arbitrary expressions, `alignof` is a common GNU extension and most compilers support it. Both Clang and GCC will warn only if `-Wpedantic` is set.

[12] In C++20, referring to the type of a generic lambda parameter explicitly is possible (due to the addition to lambdas of some familiar template syntax):

```
auto printTypeInformation = []<typename T>(T object)
{
    std::cout << "     size: " << sizeof(T) << '\n'
              << "alignment: " << alignof(T) << '\n';
};
```

[13] Note that `alignof(object)` will work on every major compiler (GCC 10.x, Clang 10.x, and MSVC 19.x) as a nonstandard extension.

see-also

See Also

- "**??**" — Safe C++11 feature that can be used to provide an artificially stricter alignment (e.g., more than **natural alignment**).
- "**??**" — Safe C++11 feature that helps work around `alignof`'s limitation of accepting only a type, not an expression (see *Annoyances* on page 12).

further-reading

Further Reading

None so far

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe
sec-unsafe-cpp11 Intro text should be here.

sec-unsafe-cpp14