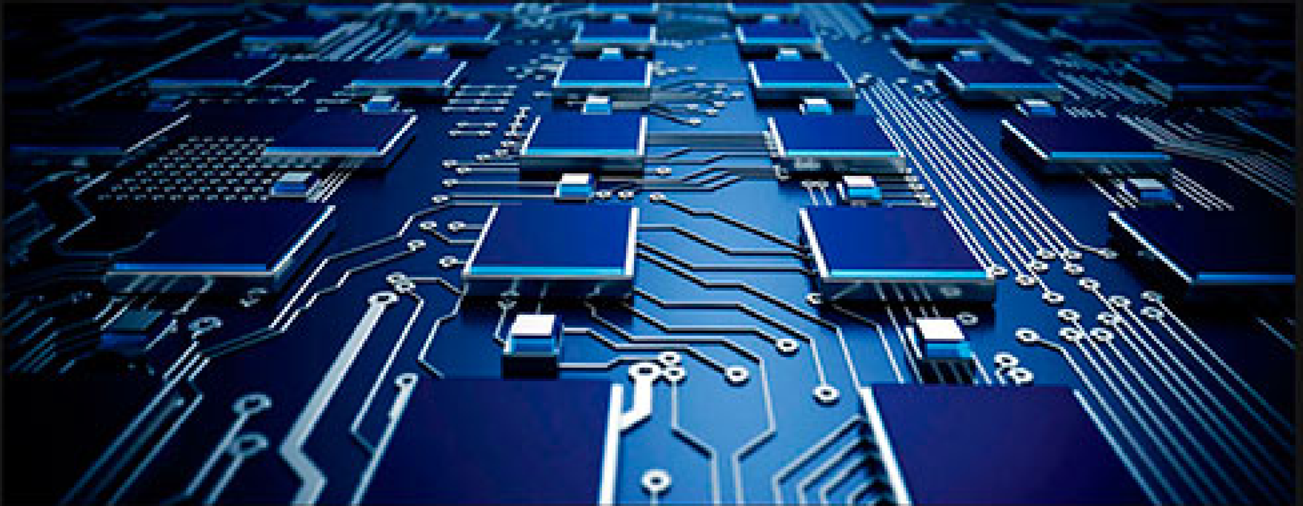
[Купить книгу C++ Memory Management]

packt



1ST EDITION

C++ Memory Management

Write leaner and safer C++ code using proven memory-management techniques

PATRICE ROY

Foreword by Michael Wong,
Distinguished Engineer, ISO C++ Standards Founding Directions Group Chair,
C++ Foundation Founding Director

1
Objects, Pointers, and References

Before we start discussing memory management in C++, let's make sure we understand each other and agree on a common vocabulary. If you're a long-time C++ programmer, you probably have your own ideas about what pointers, objects, and references are. Your ideas will stem from a wealth of experience. If you are coming to this book from another language, you might also have your own ideas as to what these terms mean in C++ and how they relate to memory and memory management.

In this chapter, we are going to make sure we have a common understanding of some basic (but profound) ideas so that we can build on this shared understanding for the rest of our adventure together. Specifically, we will explore questions such as the following:

- How is memory represented in C++? What exactly is that thing we call memory, at least in the context of the C++ language?
- What are objects, pointers, and references? What do we mean by those terms in C++? What are the lifetime rules of objects? How do they relate to memory?
- What are arrays in C++? In this language, arrays are a low-level but highly efficient construct represented in a way that directly impacts memory management.

Technical requirements

This book assumes that readers have some basic knowledge of C++ or of syntactically similar languages such as C, Java, C#, or JavaScript. For this reason, we will not explain the basics of variable declarations, loops, `if` statements, or functions.

We will, however, use some aspects of the C++ language in this chapter that some readers might be less comfortable with. Please refer to *Annexure: Things You Should Know*, before reading this book.

Some of the examples use C++20 or C++23, so make sure that your compiler supports this version of the standard to get the most out of them.

The code for this chapter can be found here: `https://github.com/PacktPublishing/C-Plus-Plus-Memory-Management/tree/main/chapter1`.

Representation of memory in C++

This is a book on memory management. You, readers, are trying to figure out what it means, and I, as the author, am trying to convey what it means.

The way in which the standard describes memory can be seen in [`wg21.link/basic.memobj`]. Essentially, memory in C++ is expressed as one or more sequences of contiguous bytes. This opens up the possibility of memory expressed as a set of discontinuous blocks of contiguous memory because, historically, C++ has supported memories made of various distinct segments. Every byte in a C++ program has a unique address.

Memory in a C++ program is populated with various entities such as objects, functions, references, and so on. Managing memory efficiently requires grasping what these entities mean and how programs can make use of them.

The meaning of the word byte is important in C++. As detailed in [`wg21.link/intro.memory`], bytes are the fundamental storage unit in C++. The number of bits in a byte is implementation-defined in C++. The standard does state, however, that a byte has to be wide enough to contain both the ordinary literal encoding of any element of the basic literal character set and the eight-bit code units of the UTF-8 encoding form. It also states that a byte is made of a contiguous sequence of bits.

What often surprises people is that in C++, a byte is not necessarily an octet: a byte consists of at least eight bits but could be made of more (something that's useful on some exotic hardware). This might change in the future, as the standard committee might constrain that definition someday, but this is the situation at the time of the publication of this book. The key idea here is that a byte is the smallest addressable unit of memory in a program.

Objects, pointers, and references

We tend to use words such as object, pointer, and reference informally, without thinking too much about what they mean. In a language such as C++, these words have precise meanings that define and delimit what we can do in practice.

Before we get our hands dirty, so to speak, let's examine the formal meaning of these terms in C++.

Objects

If we polled programmers working with different languages and asked them how they would define the term object, we could probably expect such answers as "something that groups together variables and related functions" or "an instance of a class," which correspond to traditional takes on that term from the realm of object-oriented programming.

C++ as a language tries to provide homogeneous support for user-defined types such as structs or classes. It also provides support for fundamental types such as `int` or `float`. Thus, it probably should not be surprising that, for C++, the definition of an object is expressed in terms of its properties, not in terms of what the word means, and that this definition includes the most fundamental types. The definition of an object in C++ is described in [`wg21.link/intro.object`] and takes the following factors into account:

- How the object is created explicitly, such as when defining the object or constructing it through one of the many variations of `operator new`. The object may also be created implicitly such as when creating a temporary object as the result of some expression or when changing the active member of a `union`.
- The fact that an object is somewhere (it has an address) and occupies a region of storage of non-zero size, from the start of its construction to the end of its destruction.
- Other properties of an object, including its name (if it has one), its type, and its storage duration (`automatic`, `static`, `thread_local`, and so on.).

The C++ standard explicitly calls out functions as not being objects, even if a function has an address and occupies storage.

From this, we can infer that even a humble `int` is an object, but a function is not. You can see already, dear reader, that the book you're reading will touch on fundamental topics, since lifetime and the storage occupied by objects are part of the fundamental properties of these entities we use in our programs every day. Such things as lifetime and storage are clearly part of what memory management is about. You can convince yourself of that fact with this simple program:

```
#include <type_traits>
int main() {
   static_assert(std::is_object_v<int>);
   static_assert(!std::is_object_v<decltype(main)>);
}
```

What is an object? It's something that has a lifetime and occupies storage. Controlling these characteristics is part of the reasons why this book exists.

Pointers

There are numerous (around 2,000) mentions of the word "pointer" in the text of the C++ standard, but if you open an electronic copy of that document and search through it, you'll find that a formal definition is surprisingly hard to come by. This can be surprising given the fact that people tend to associate that idea with C and (by extension) C++.

Let us try to offer a useful yet informal definition, then: a pointer is a typed address. It associates a type with what is found at some location in memory. For that reason, in code like the following, one reads that `n` is an `int` object and that `p` points to an `int` object that happens to be the address of the `n` object:

```
int n = 3; // n is an int object
char c;
// int *p = &c; // no, illegal
int *p = &n;
```

It's important to understand here that `p` indeed points to an `int`, unless `p` is left uninitialized, `p` points to `nullptr`, or programmers have played tricks with the type system and made `p` point to something else deliberately. Of course, pointer `p` is an object, as it respects all the rules to that effect.

Much of the (syntactic) confusion about pointers probably comes from the contextual meaning of the `*` and `&` symbols. The trick is to remember that they have different roles when they appear in the introduction of a name and when they are used on an existing object:

```
int m = 4, n = 3;
int *p; // p declares (and defines) a pointer to an int
        // (currently uninitialized), introducing a name
p = 0; // p is a null pointer (it does not necessarily
       // point to address zero; 0 as used here is
       // just a convention)
p = nullptr; // likewise, but clearer. Prefer nullptr to
             // literal 0 whenever possible to describe
             // a null pointer
p = &m; // p points to m (p contains the address of m)
assert(*p == 4); // p already exists; with *p we are
                 // accessing what p points to
p = &n; // p now points to n (p contains the address of n)
int *q = &n; // q declares (and defines) a pointer to an
             // int and &n represents the address of n, the
             // address of an int: q is a pointer to an int
assert(*q == 3); // n holds 3 at this stage, and q points
                 // to n, so what q points to has value 3
assert(*p == 3); // the same holds for p
assert(p == q); // p and q point to the same int object
*q = 4; // q already exists, so *q means "whatever q
        // points to"
assert(n == 4); // indeed, n now holds value 4 since we
                // modified it indirectly through q
auto qq = &q;   // qq is the address of q, and its type is
                // "pointer to a pointer to an int", thus
```

```
                  // int **... But we will rarely - if ever -
                  // need this
int &r = n; // declaration of r as a reference to integer n
            // (see below). Note that & is used in a
            // declaration in this case
```

As you can see, when introducing an object, `*` means "pointer to." On an existing object, it means "what that pointer points to" (the pointee). Similarly, when introducing a name, `&` means "reference to" (something we will discuss imminently). On an existing object, it means "address of" and yields a pointer.

Pointers allow us to do arithmetic, but that's (legitimately) seen as a dangerous operation, as it can take us to arbitrary locations in a program and can therefore lead to serious damage. Arithmetic on a pointer depends on its type:

```
int *f();
char *g();
int danger() {
   auto p = f(); // p points to whatever f() returned
   int *q = p + 3; // q points to where p points to plus
                   // three times the size of an int. No
                   // clue where this is, but it's a bad,
                   // bad idea...
   auto pc = g(); // pc points to whatever g() returned
   char * qc = pc + 3; // qc points to where pc points
                       // to plus three times the size
                       // of a char. Please don't make
                       // your pointers go to places you
                       // don't know about like this
}
```

Of course, accessing the contents of arbitrary addresses is just asking for trouble. This is because it would mean invoking undefined behavior (described in *Chapter 2*), and if you do that, you're on your own. Please do not do such things in real code, as you could hurt programs – or worse, people. C++ is powerful and flexible, but if you program in C++, you're expected to behave responsibly and professionally.

C++ has four special types for pointer manipulation:

- `void*` means "address with no specific (type-related) semantics." A `void*` is an address with no associated type. All pointers (if we discount the `const` and `volatile` qualifiers) are implicitly convertible to `void*`; an informal way to read this is as "all pointers, regardless of type, really are addresses." The converse does not hold. For example, it's not true that all addresses are implicitly convertible to `int` pointers.

- `char*` means "pointer to a byte." Due to the C language roots of C++, a `char*` can alias any address in memory (the `char` type, regardless of its name, which evocates "character", really means "byte" in C and, by extension, in C++). There is an ongoing effort in C++ to give `char` the meaning of "character," but as of this writing, a `char*` can alias pretty much anything in a program. This hampers some compiler optimization opportunities (it is hard to constrain or reason about something that can lead to literally anything in memory).
- `std::byte*` is the new "pointer to a byte," at least since C++17. The (long-term) intent of `byte*` is to replace `char*` in those functions that do byte-per-byte manipulation or addressing, but since there's so much code that uses `char*` to that effect, this will take time.

For an example of conversion from and to `void*`, consider the following:

```
int n = 3;
int *p = &n; // fine so far
void *pv = p; // Ok, a pointer is an address
// p = pv; // no, a void* does not necessarily point to
           // an int (Ok in C, not in C++)
p = static_cast<int *>(pv); // fine, you asked for it, but
                            // if you're wrong you're on
                            // your own
```

The following example, which is somewhat more elaborate, uses `const char*` (but could use `const byte*` instead). It shows that one can compare the byte-per-byte representation of two objects, at least in some circumstances, to see whether they are equivalent:

```
#include <iostream>
#include <type_traits>
using namespace std;
bool same_bytes(const char *p0, const char *p1,
                std::size_t n) {
    for(std::size_t i = 0; i != n; ++i)
        if(*(p0 + i) != *(p1 + i))
            return false;
    return true;
}
template <class T, class U>
   bool same_bytes(const T &a, const U &b) {
      using namespace std;
       static_assert(sizeof a == sizeof b);
       static_assert(has_unique_object_representations_v<
          T
       >);
       static_assert(has_unique_object_representations_v<
          U
```

```
        >);
        return same_bytes(reinterpret_cast<const char*>(&a),
                          reinterpret_cast<const char*>(&b),
                          sizeof a);
    }
struct X {
   int x {2}, y{3};
};
struct Y {
   int x {2}, y{3};
};
#include <cassert>
int main() {
    constexpr X x;
    constexpr Y y;
    assert(same_bytes(x, y));
}
```

The `has_unique_object_representations` trait is true for types uniquely defined by their values, that is, types exempt of padding bits.. That's sometimes important as C++ does not say what happens to padding bits in an object, and performing a bit-per-bit comparison of two objects might yield surprising results. Note that objects of floating point types are not considered uniquely defined by their values as there are many distinct values that qualify as NaN, or "not a number".

References

The C++ language supports two related families of indirections: pointers and references. Like their cousins, the pointers, references are often mentioned by the C++ standard (more than 1,800 times) but it's hard to find a formal definition for them.

We will try once again to provide an informal but operational definition: a reference can be seen as an alias for an existing entity. We deliberately did not use object, since one could refer to a function and we already know that a function is not an object.

Pointers are objects. As such, they occupy storage. References, on the other hand, are not objects and use no storage of their own, even though an implementation could simulate their existence with pointers. Compare `std::is_object_v<int*>` with `std::is_object_v<int&>`: the former is `true`, and the latter is `false`.

The `sizeof` operator, applied to a reference, will yield the size of what it refers to. Consequently, taking the address of a reference yields the address of what it refers to.

In C++, a reference is always bound to an object and remains bound to that object until the end of the reference's lifetime. A pointer, on the other hand, can point to numerous distinct objects during its lifetime, as we have seen before:

```
// int &nope; // would not compile (what would nope
              // refer to?)
int n = 3;
int &r = n; // r refers to n
++r; // n becomes 4
assert(&r == &n); // taking the address of r means taking
                  // the address of n
```

Another difference between pointers and references is that, contrary to the situation that prevails with pointers, there is no such thing as reference arithmetic. This makes references somewhat safer than pointers. There is room for both kinds of indirections in a program (and we will use them both in this book!), but for everyday programming, a good rule of thumb is to use references if possible and to use pointers if necessary.

Now that we have examined the representation of memory and taken a look at how C++ defines some fundamental ideas such as a byte, an object, a pointer, or a reference, we can delve a little deeper into some important defining properties of objects.

Understanding the fundamental properties of objects

We saw earlier that in C++, an object has a type and an address. It also occupies a region of storage from the beginning of its construction to the end of its destruction. We will now examine these fundamental properties in more detail in order to understand how these properties affect the ways in which we write programs.

Object lifetime

One of C++'s strengths, but also one reason for its relative complexity, arises from the control one has over the lifetime of objects. In C++, generally speaking, automatic objects are destructed at the end of their scope in a well-defined order. Static (global) objects are destructed on program termination in a somewhat well-defined order (in a given file, the order of destruction is clear, but it's more complicated for static objects in different files). Dynamically allocated objects are destructed "when your program says so" (there are many nuances to this).

Let's examine some aspects of object lifetime with the following (very) simple program:

```
#include <string>
#include <iostream>
#include <format>
struct X {
```

```
    std::string s;
    X(std::string_view s) : s{ s } {
        std::cout << std::format("X::X({})\n", s);
    }
    ~X(){
        std::cout << std::format("~X::X() for {}\n", s);
    }
};
X glob { "glob" };
void g() {
    X xg{ «g()» };
}
int main() {
    X *p0 = new X{ "p0" };
    [[maybe_unused]] X *p1 = new X{ "p1" }; // will leak
    X xmain{ "main()" };
    g();
    delete p0;
    // oops, forgot delete p1
}
```

When executed, that program will print the following:

```
X::X(glob)
X::X(p0)
X::X(p1)
X::X(main())
X::X(g())
~X::X() for g()
~X::X() for p0
~X::X() for main()
~X::X() for glob
```

The fact that the number of constructors and destructors do not match is a sign that we did something wrong. More specifically, in this example, we manually created an object (pointed to by `p1`) with `operator new` but never manually destructed that object afterward.

One common source of confusion for programmers unfamiliar with C++ is the distinction between pointer and pointee. In this program, `p0` and `p1` are both destructed when reaching the end of their scope (by the closing brace of the `main()` function), just as `xmain` will be. However, since `p0` and `p1` point to dynamically allocated objects, the pointees have to be explicitly destructed, something we did for `p0` but (deliberately, for the sake of the example) neglected to do for `p1`.

What happens to p1's pointee then? Well, it has been manually constructed and has not been manually destructed. As such, it floats in memory where no one can access it anymore. This is what people often call a memory leak: a chunk of memory your program allocated but never deallocated.

Worse than leaking the storage for the X object pointed to by p1, however, is the fact that the pointee's destructor will never be called, which can cause all sorts of resource leaks (files not closed, database connections not closed, system handles not released, and so on). In *Chapter 4*, *Using Destructors*, we will examine how it is possible to avoid such situations and write clean, simple code at the same time.

Object size, alignment, and padding

Since each object occupies storage, the space associated with an object is an important (if low-level) property of C++ types. For example, look at the following code:

```
class B; // forward declaration: there will be a class B
         // at some point in the future
void f(B*); // fine, we know what B is, even if we don't
            // know the details yet, and all object
            // addresses are of the same size
// class D : B {}; // oops! To know what a D is, we have
                   // to know how big a B is and what a
                   // B object contains since a D is a B
```

In that example, trying to define the D class would not compile. This is because in order to create a D object, the compiler needs to reserve enough space for a D object, but a D object is also a B object, and as such we cannot know the size of a D object without knowing the size of a B object.

The size of an object or, equivalently, of a type can be obtained through the sizeof operator. This operator yields a compile-time, non-zero unsigned integral value corresponding to the number of bytes required to store an object:

```
char c;
// a char occupies precisely one byte of storage, per
// standard wording
static_assert(sizeof c == 1); // for objects parentheses
                              // are not required
static_assert(sizeof(c) == 1); // ... but you can use them
static_assert(sizeof(char) == 1); // for types, parentheses
                                  // are required
struct Tiny {};
// all C++ types occupy non-zero bytes of storage by
// definition, even if they are "empty" like type Tiny
static_assert(sizeof(Tiny) > 0);
```

In the preceding example, the `Tiny` class is empty because it has no data member. A class could have member functions and still be empty. Empty classes that expose member functions are very commonly used in C++.

A C++ object always occupies at least one byte of storage, even in the case of empty classes such as `Tiny`. That's because if an object's size was zero, that object could be at the same memory location as its immediate neighbor, which would be somewhat hard to reason about.

C++ differs from many other languages in that it does not standardize the size of all fundamental types. For example, `sizeof(int)` can yield different values depending on the compiler and platform. Still, there are rules concerning the size of objects:

- The size reported by operator `sizeof` for objects of type `signed char`, `unsigned char` and `char` is 1, and the same goes for `sizeof(std::byte)` as each of these types can be used to represent a single byte.
- Expressions `sizeof(short)>=sizeof(char)` and `sizeof(int)>=sizeof(short)` will hold on all platforms, which means that there might be cases where `sizeof(char)` and `sizeof(int)` are both 1. In terms of width (i.e., bits used in the value representation) of fundamental types, the C++ standard limits itself to stating the minimum width for each type. The list can be found at [`wg21.link/tab:basic.fundamental.width`].
- As we have already said, expression `sizeof(T)>0` holds for any type `T`. In C++, there are no zero-sized objects, not even objects of empty classes.
- The size occupied by an object of any `struct` or `class` cannot be less than the sum of the size of its data members (but there are caveats).

This last rule deserves an explanation. Consider the following situation:

```
class X {};
class Y {
   X x;
};
int main() {
   static_assert(sizeof(X) > 0);
   static_assert(sizeof(Y) == sizeof(X)); // <-- here
}
```

The line marked `<-- here` might be intriguing. Why would `sizeof(Y)` be equal to `sizeof(X)` if every `Y` object contains an `X` object? Remember that `sizeof(X)` is greater than `0` even though `X` is an empty class because every C++ object has to occupy at least one byte of storage. However, in the case of `Y`, which is not an empty class, each `Y` object already occupies storage due to its `x` data member. There's no reason to somewhat artificially add storage space to objects of that type.

Now, consider this:

```
class X {
   char c;
};
class Y {
   X x;
};
int main() {
   static_assert(sizeof(X) == sizeof(char)); // <-- here
   static_assert(sizeof(Y) == sizeof(X)); // <-- here too
}
```

The same reasoning applies again: an object of type `X` occupies the same amount of storage space as its only data member (of type `char`), and an object of type `Y` occupies the same amount of storage space as its only data member (of type `X`).

Continuing this exploration, consider this :

```
class X { };
class Y {
   X x;
   char c;
};
int main() {
   static_assert(sizeof(Y) >= sizeof(char) + sizeof(X));
}
```

This is the rule we mentioned earlier but expressed formally for a specific type. In this situation, supposing that `sizeof(X)` being equal to `1` is highly probable, one could even reasonably expect that `sizeof(Y)` would be equal to the sum of `sizeof(char)` and `sizeof(X)`.

Finally, consider this:

```
class X { };
class Y : X { // <--  private inheritance
   char c;
};
int main() {
   static_assert(sizeof(Y) == sizeof(char)); // <-- here
}
```

We moved from having an object of class X being a data member of Y to X being a base class of Y. This has an interesting consequence: since the base class X is empty, and since we know from definition that objects of the derived class Y will occupy at least one byte of storage, the base class can be flattened into the derived class for Y objects. This is a useful optimization called the **empty base optimization**. You can reasonably expect compilers to perform this optimization in practice, at least in the case of single inheritance relationships.

Note that since the presence of an X in a Y is an implementation detail, not something that participates in the interface of class Y, we used private inheritance in this example. The empty base optimization would apply with public or protected inheritance too, but in this case, private inheritance preserves the fact that the X part of a Y is something that only the Y knows about.

Since C++20, if you think composition would be more appropriate than inheritance to describe the relation between two classes such as X and Y, you can mark a data member as `[[no_unique_address]]` to inform the compiler that this member, if it is an object of an empty class, does not have to occupy storage within the enclosing object. Compilers are not forced to comply, since attributes can be ignored, so make sure to verify that your chosen compilers implement this before writing code that relies on this:

```
class X { };
class Y {
   char c;
   [[no_unique_address]] X x;
};
int main() {
   static_assert(sizeof(X) > 0);
   static_assert(sizeof(Y) == sizeof(char)); // <-- here
}
```

All of the examples so far have been very simple, using classes with zero, one, or two very small data members. Code is rarely so simple. Consider the following program:

```
class X {
   char c; // sizeof(char) == 1 by definition
   short s;
   int n;
};
int main() {
   static_assert(sizeof(short) == 2); // we suppose this...
   static_assert(sizeof(int) == 4);   // ... and this
   static_assert(
      sizeof(X) >= sizeof(char)+sizeof(short)+sizeof(int)
   );
}
```

Supposing that the first two static assertions hold, which is probable but not guaranteed, we know that `sizeof(X)` will be at least `7` (the sum of the sizes of its data members). In practice, however, you will probably see that `sizeof(X)` is equal to 8. Now, this might seem surprising at first, but it's a logical consequence of something called **alignment**.

The alignment of an object (or of its type) tells us where that object can be placed in memory. The `char` type has an alignment of `1`, and as such one can place a `char` object literally anywhere (as long as one can access that memory). For an alignment of `2` (which is likely for type `short`), objects can only be placed at addresses that are a multiple of 2. More generally, if a type has an alignment of `n`, then objects of that type must be placed at an address that is a multiple of `n`. Note that alignment has to be a strictly positive power of 2; not respecting this rule incurs undefined behavior. Of course, your compiler will not put you in that position, but you might put yourself in such trouble if you're not careful, given some of the tricks we will be using in this book. With great control comes great responsibility.

The C++ language offers two operators related to alignment:

- The `alignof` operator, which yields the natural alignment of a type `T` or of an object of that type.
- The `alignas` operator, which lets programmers impose the alignment of an object. This is often useful when playing tricks with memory (as we will) or when interfacing with exotic hardware (the term "exotic" here can be taken in a very broad sense). Of course, `alignas` can only reasonably increase the natural alignment of a type, not reduce it.

For some fundamental type `T`, one can expect the assertion that `sizeof(T)` is equal to `alignof(T)` to hold, but that assertion does not generalize to composite types. For example, consider the following:

```
class X {
   char c;
   short s;
   int n;
};
int main() {
   static_assert(sizeof(short) == alignof(short));
   static_assert(sizeof(int) == alignof(int));
   static_assert(sizeof(X) == 8); // highly probable
   static_assert(alignof(X) == alignof(int)); // likewise
}
```

Generally speaking, for a composite type, the alignment will correspond to the worst alignment of its data members. Here, "worst" means "biggest." For class `X`, the worst-aligned data member is `n` of type `int` and as such, `X` objects will be aligned on boundaries of `alignof(int)` bytes.

You might wonder now why we can expect the assertion that `sizeof(X)` is equal to `8` to hold if `sizeof(short)==2` and `sizeof(int)==4`. Let's look at the probable layout for objects of the `X` type:

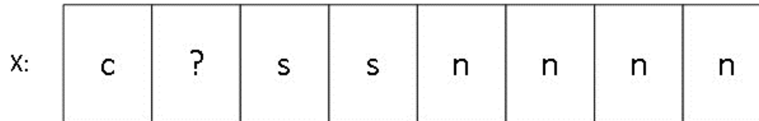


Figure 1.1 – Compact layout of an object of type X in memory

Each box in this figure is a byte in memory. As we can see, there's a `?` between `c` and the first byte of `s`. That comes from alignment. If `alignof(short)==2` and `alignof(int)==4`, then the only correct layout for an `X` object places its `n` member at a boundary of `4`. This means that there will be a padding byte (a byte that does not participate in the value representation of `X`) between `c` and `s` to align `s` on a two-byte boundary and to align `n` on a four-byte boundary.

What might seem more surprising is that the order in which data members are laid out in a class impacts the size of the objects of that class. For example, consider the following:

```
class X {
   short s;
   int n;
   char c;
};
int main() {
   static_assert(sizeof(short) == alignof(short));
   static_assert(sizeof(int) == alignof(int));
   static_assert(alignof(X) == alignof(int));
   static_assert(sizeof(X) == 12); // highly probable
}
```

That often surprises people, but it's true, and something to think about. With this example, the probable layout for an `X` object would be as follows:

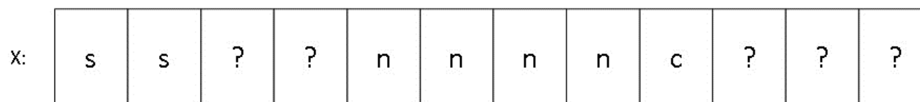


Figure 1.2 – Less compact layout for an object of type X in memory

By now, the two `?` "squares" between `s` and `n` are probably clear, but the three trailing `?` "squares" might seem surprising. After all, why add padding at the end of an object?

The answer is *because of arrays*. As we will soon discuss, elements of an array are contiguous in memory, and for that reason, it is important that each element of an array is properly aligned. In a case such as this, the trailing padding bytes in an object of class `X` ensure that if an element in an array of `X` objects is properly aligned, then the next element will be properly aligned too.

Now that you know about alignment, consider that just changing the order of elements from one version of class `X` to another resulted in a memory consumption increase of 50% for each object of that type. That hurts your program's memory space consumption and its speed all at once. C++ compilers cannot reorder your data members for you, as your code sees the addresses of objects. Changing the relative position of data members could break users' code, so it's up to programmers to be careful with their chosen layouts. Note that keeping objects small is not the only factor that can influence the choice of layout in an object, especially in multithreaded code (where sometimes keeping two objects at a distance from one another can lead to better cache usage), so one should remember that layout is important, but not something to take on naïvely.

Copy and movement

At this point, we need to say a few words about copy and movement, two fundamental considerations in a language such as C++ where there are actual objects.

The C++ language considers six member functions as special. These functions will be automatically generated for your types unless you take steps to prevent it. These are as follows:

- **The default constructor**: It's probably the least special of all six, as it's only implicitly generated if you write no constructor of your own.
- **The destructor**: This is called at the end of an object's lifetime.
- **The copy constructor**: It is called when constructing an object with a single object of the same type as argument.
- **The copy assignment**: It is called when replacing the contents of an existing object with a copy of the contents of another object.
- **The move constructor**: It is called when constructing an object from a reference to an object one can move from. Examples of movable-from objects include objects one could not refer to anymore, such as the (anonymous) result of evaluating an expression or one being returned by a function. The program can also explicitly make an object movable-from with `std::move()`.
- **The move assignment**: It behaves like copy assignment but is applied when the argument passed to the assignment operator is something one can move from.