3

Fundamentals of Software Engineering for Games

In this chapter, we'll briefly review the basic concepts of object-oriented programming (OOP) and then delve into some advanced topics that should prove invaluable in any software engineering endeavor (and especially when creating games). As with Chapter 2, I hope you will not to skip this chapter entirely; it's important that we all embark on our journey with the same set of tools and supplies.

3.1 C++ Review and Best Practices

Because C++ is arguably the most commonly used language in the game industry, we will focus primarily on C++ in this book. However, most of the concepts we'll cover apply equally well to *any* object-oriented programming language. Certainly a great many other languages are used in the game industry—imperative languages like C; object-oriented languages like C# and Java; scripting languages like Python, Lua and Perl; functional languages like Lisp, Scheme and F#, and the list goes on. I highly recommend that every programmer learn at least two high-level languages (the more the merrier), as

well as learning at least some *assembly language* programming. Every new language that you learn further expands your horizons and allows you to think in a more profound and proficient way about programming overall. That being said, let's turn our attention now to object-oriented programming concepts in general, and C++ in particular.

3.1.1 Brief Review of Object-Oriented Programming

Much of what we'll discuss in this book assumes you have a solid understanding of the principles of object-oriented design. If you're a bit rusty, the following section should serve as a pleasant and quick review. If you have no idea what I'm talking about in this section, I recommend you pick up a book or two on object-oriented programming (e.g., [5]) and C++ in particular (e.g., [41] and [31]) before continuing.

3.1.1.1 Classes and Objects

A *class* is a collection of attributes (data) and behaviors (code) that together form a useful, meaningful whole. A class is a *specification* describing how individual *instances* of the class, known as *objects*, should be constructed. For example, your pet Rover is an instance of the class "dog." Thus, there is a one-to-many relationship between a class and its instances.

3.1.1.2 Encapsulation

Encapsulation means that an object presents only a limited interface to the outside world; the object's internal state and implementation details are kept hidden. Encapsulation simplifies life for the user of the class, because he or she need only understand the class' limited interface, not the potentially intricate details of its implementation. It also allows the programmer who wrote the class to ensure that its instances are always in a logically consistent state.

3.1.1.3 Inheritance

Inheritance allows new classes to be defined as *extensions* to preexisting classes. The new class modifies or extends the data, interface and/or behavior of the existing class. If class `Child` extends class `Parent`, we say that `Child` *inherits from* or is *derived from* `Parent`. In this relationship, the class `Parent` is known as the *base class* or *superclass*, and the class `Child` is the *derived class* or *subclass*. Clearly, inheritance leads to hierarchical (tree-structured) relationships between classes.

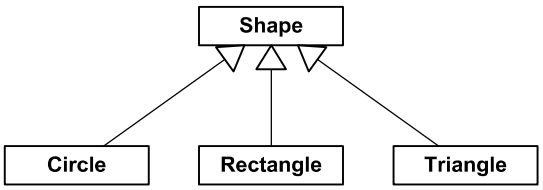

Figure 3.1. UML static class diagram depicting a simple class hierarchy.

Inheritance creates an "is-a" relationship between classes. For example, a circle *is a* type of shape. So, if we were writing a 2D drawing application, it would probably make sense to derive our `Circle` class from a base class called `Shape`.

We can draw diagrams of class hierarchies using the conventions defined by the Unified Modeling Language (UML). In this notation, a rectangle represents a class, and an arrow with a hollow triangular head represents inheritance. The inheritance arrow points from child class to parent. See Figure 3.1 for an example of a simple class hierarchy represented as a UML *static class diagram.*

Multiple Inheritance

Some languages support *multiple inheritance* (MI), meaning that a class can have more than one parent class. In theory MI can be quite elegant, but in practice this kind of design usually gives rise to a lot of confusion and technical difficulties (see http://en.wikipedia.org/wiki/Multiple_inheritance). This is because multiple inheritance transforms a simple *tree* of classes into a potentially complex *graph.* A class graph can have all sorts of problems that never plague a simple tree—for example, the *deadly diamond* (http://en.wikipedia.org/wiki/Diamond_problem), in which a derived class ends up containing *two copies* of a grandparent base class (see Figure 3.2). (In C++, *virtual inheritance* allows one to avoid this doubling of the grandparent's data.) Multiple inheritance also complicates casting, because the actual address of a pointer may change depending on which base class it is cast to. This happens because of the presence of multiple vtable pointers within the object.

Most C++ software developers avoid multiple inheritance completely or only permit it in a limited form. A common rule of thumb is to allow only simple, parentless classes to be multiply inherited into an otherwise strictly single-inheritance hierarchy. Such classes are sometimes called *mix-in classes* because they can be used to introduce new functionality at arbitrary points in a class tree. See Figure 3.3 for a somewhat contrived example of a mix-in class.

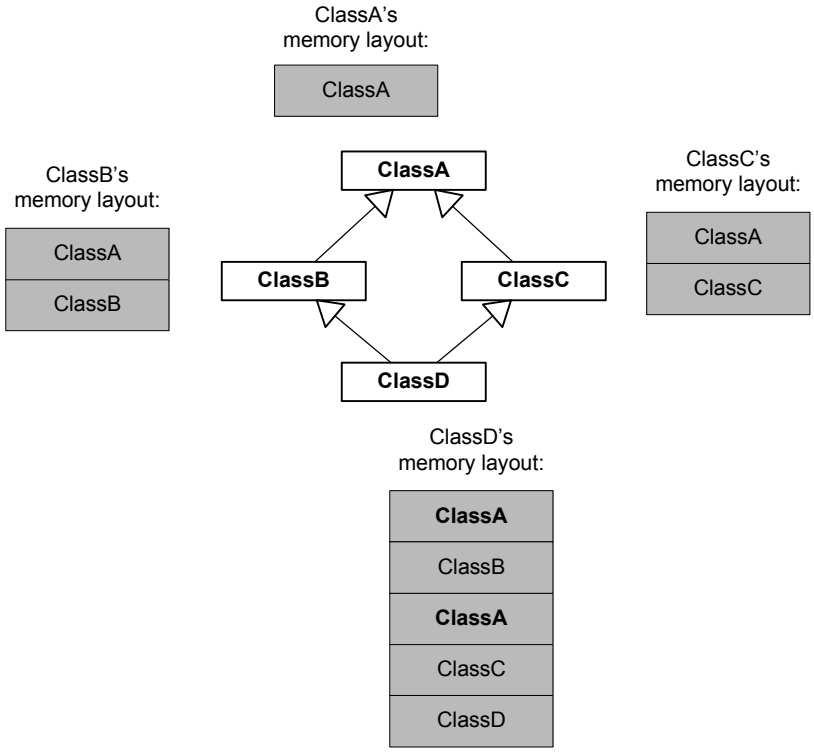

Figure 3.2. "Deadly diamond" in a multiple inheritance hierarchy.

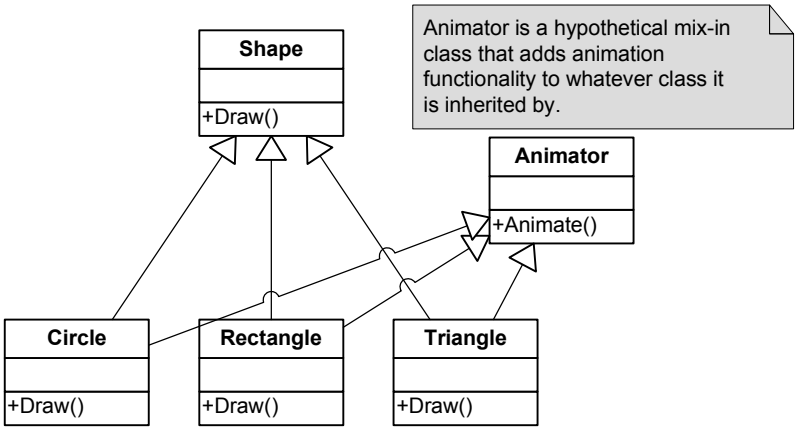

Figure 3.3. Example of a mix-in class.

3.1.1.4 Polymorphism

Polymorphism is a language feature that allows a collection of objects of different types to be manipulated through a single *common interface*. The common interface makes a heterogeneous collection of objects *appear* to be homogeneous, from the point of view of the code using the interface.

For example, a 2D painting program might be given a list of various shapes to draw on-screen. One way to draw this heterogeneous collection of shapes is to use a switch statement to perform different drawing commands for each distinct type of shape.

```
void drawShapes(std::list<Shape*> shapes)
{
   std::list<Shape*>::iterator pShape = shapes.begin();
   std::list<Shape*>::iterator pEnd = shapes.end();

   for ( ; pShape != pEnd; pShape++)
   {
      switch (pShape->mType)
      {
      case CIRCLE:
         // draw shape as a circle
         break;

      case RECTANGLE:
         // draw shape as a rectangle
         break;

      case TRIANGLE:
         // draw shape as a triangle
         break;

      //...
      }
   }
}
```

The problem with this approach is that the `drawShapes()` function needs to "know" about all of the kinds of shapes that can be drawn. This is fine in a simple example, but as our code grows in size and complexity, it can become difficult to add new types of shapes to the system. Whenever a new shape type is added, one must find every place in the code base where knowledge of the set of shape types is embedded—like this switch statement—and add a case to handle the new type.

The solution is to insulate the majority of our code from any knowledge of the types of objects with which it might be dealing. To accomplish this, we can define classes for each of the types of shapes we wish to support. All of these classes would inherit from the common base class `Shape`. A *virtual function*—the C++ language's primary polymorphism mechanism—would be defined called `Draw()`, and each distinct shape class would implement this function in a different way. Without "knowing" what specific types of shapes it has been given, the drawing function can now simply call each shape's `Draw()` function in turn.

```
struct Shape
{
   virtual void Draw() = 0; // pure virtual function
};

struct Circle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a circle
   }
};

struct Rectangle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a rectangle
   }
};

struct Triangle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a triangle
   }
};

void drawShapes(std::list<Shape*> shapes)
{
   std::list<Shape*>::iterator pShape = shapes.begin();
   std::list<Shape*>::iterator pEnd = shapes.end();

   for ( ; pShape != pEnd; pShape++)
   {
```

```
            pShape->Draw(); // call virtual function
        }
    }
```

3.1.1.5 Composition and Aggregation

Composition is the practice of using a *group* of *interacting* objects to accomplish a high-level task. Composition creates a "has-a" or "uses-a" relationship between classes. (Technically speaking, the "has-a" relationship is called *composition*, while the "uses-a" relationship is called *aggregation*.) For example, a spaceship *has an* engine, which in turn *has a* fuel tank. Composition/aggregation usually results in the individual classes being simpler and more focused. Inexperienced object-oriented programmers often rely too heavily on inheritance and tend to underutilize aggregation and composition.

As an example, imagine that we are designing a graphical user interface for our game's front end. We have a class `Window` that represents any rectangular GUI element. We also have a class called `Rectangle` that encapsulates the mathematical concept of a rectangle. A naïve programmer might derive the `Window` class from the `Rectangle` class (using an "is-a" relationship). But in a more flexible and well-encapsulated design, the `Window` class would *refer to* or *contain* a `Rectangle` (employing a "has-a" or "uses-a" relationship). This makes both classes simpler and more focused and allows the classes to be more easily tested, debugged and reused.

3.1.1.6 Design Patterns

When the same type of problem arises over and over, and many different programmers employ a very similar solution to that problem, we say that a *design pattern* has arisen. In object-oriented programming, a number of common design patterns have been identified and described by various authors. The most well-known book on this topic is probably the "Gang of Four" book [17].

Here are a few examples of common general-purpose design patterns.

- *Singleton.* This pattern ensures that a particular class has only one instance (the *singleton instance*) and provides a global point of access to it.
- *Iterator.* An iterator provides an efficient means of accessing the individual elements of a collection, without exposing the collection's underlying implementation. The iterator "knows" the implementation details of the collection so that its users don't have to.
- *Abstract factory.* An abstract factory provides an interface for creating families of related or dependent classes without specifying their concrete classes.

The game industry has its own set of design patterns for addressing problems in every realm from rendering to collision to animation to audio. In a sense, this book is all about the high-level design patterns prevalent in modern 3D game engine design.

3.1.2 Coding Standards: Why and How Much?

Discussions of coding conventions among engineers can often lead to heated "religious" debates. I do not wish to spark any such debate here, but I will go so far as to suggest that following at least a minimal set of coding standards is a good idea. Coding standards exist for two primary reasons.

1. Some standards make the code more readable, understandable and maintainable.
2. Other conventions help to prevent programmers from shooting themselves in the foot. For example, a coding standard might encourage the programmer to use only a smaller, more testable and less error-prone subset of the whole language. The C++ language is rife with possibilities for abuse, so this kind of coding standard is particularly important when using C++.

In my opinion, the most important things to achieve in your coding conventions are the following.

- *Interfaces are king.* Keep your interfaces (.h files) clean, simple, minimal, easy to understand and well-commented.
- *Good names encourage understanding and avoid confusion.* Stick to intuitive names that map directly to the purpose of the class, function or variable in question. Spend time up-front identifying a good name. Avoid a naming scheme that requires programmers to use a look-up table in order to decipher the meaning of your code. Remember that high-level programming languages like C++ are intended for *humans* to read. (If you disagree, just ask yourself why you don't write all your software directly in machine language.)
- *Don't clutter the global namespace.* Use C++ namespaces or a common naming prefix to ensure that your symbols don't collide with symbols in other libraries. (But be careful not to overuse namespaces, or nest them too deeply.) Name `#define`d symbols with extra care; remember that C++ preprocessor macros are really just text substitutions, so they cut across all C/C++ scope and namespace boundaries.
- *Follow C++ best practices.* Books like the *Effective C++* series by Scott Meyers [31,32], Meyers' *Effective STL* [33] and *Large-Scale C++ Software Design*

by John Lakos [27] provide excellent guidelines that will help keep you out of trouble.

- *Be consistent.* The rule I try to use is as follows: If you're writing a body of code from scratch, feel free to invent any convention you like—then stick to it. When editing preexisting code, try to follow whatever conventions have already been established.
- *Make errors stick out.* Joel Spolsky wrote an excellent article on coding conventions, which can be found at http://www.joelonsoftware.com/articles/Wrong.html. Joel suggests that the "cleanest" code is not necessarily code that looks neat and tidy on a superficial level, but rather the code that is written in a way that makes common programming errors *easier to see*. Joel's articles are always fun and educational, and I highly recommend this one.

3.1.3 C++11

C++11 is the most-recent variant of the C++ programming language standard. It was approved by the ISO on August 12, 2011, replacing C++03 (which itself replaced the first standardized version of the language, C++98). C++11 was formerly known as C++0x.

C++11 introduces a number of new powerful language features. There are plenty of great online resources and books that describe these features in detail, so we won't attempt to cover them here. Instead, we'll just survey the key features to serve as a jumping-off point for further reading. However, we will cover *move semantics* in some depth because the concepts are a bit tricky to understand.

3.1.3.1 `auto`

The `auto` keyword is not new to the C++ language, but its meaning has changed for C++11. In C++03 it is a *storage class specifier*, along with `static`, `register` and `extern`. Only one of these four specifiers can be used on a given varaible, but the default storage class is `auto`, meaning that the variable has local scope and should be allocated in a register (if one is available) or else on the program stack. In C++11, the `auto` keyword is now used for variable type inference, meaning it can be used *in place of* a type specifier—the compiler infers the type from the right-hand side of the variable's initializer expression.

```
// C++03
float f = 3.141592f;
__m128 acc = _mm_setzero_ps();
```

```
std::map<std::string, std::int32_t>::const_iterator it
                                        = myMap.begin();

// C++11
auto f = 3.141592f;
auto acc = _mm_setzero_ps();
auto it = myMap.begin();
```

3.1.3.2 `nullptr`

In prior versions of C and C++, a `NULL` pointer was specified by using the literal `0`, sometimes cast to `(void*)` or `(char*)`. This lacked type safety and could cause problems because of C/C++'s implicit integer conversions. C++11 introduces the type-safe explicit literal value `nullptr` to represent a null pointer; it is an instance of the type `std::nullptr_t`.

3.1.3.3 Range-Based `for` Loops

C++11 extends the `for` statement to support a short-hand "foreach" declaration style. This allows you to iterate over C-style arrays and any other data structure for which the non-member `begin()` and `end()` functions are defined.

```
// C++03
for (std::map<std::string, std::int32_t>::const_iterator it
         = myMap.begin();
     it != myMap.end();
     it++)
{
    printf("%s\n", it->first.c_str());
}

// C++11
for (const auto& pair : myMap)
{
    printf("%s\n", pair.first.c_str());
}
```

3.1.3.4 `override` and `final`

The `virtual` keyword in C++ can lead to confusing and possibly erroneous code, because the language makes no distinction between:

- introducing a *new* virtual function into a class,
- *overriding* an inherited virtual function, and
- implementing a *leaf* virtual function that is not intended to be overridden by derived classes.

Also, C++ does not require the programmer to use the `virtual` keyword on overridden virtual functions at all. To partially rectify this state of affairs, C++11 introduces two new identifiers which can be tacked on to the end of virtual function declarations, thereby making the programmer's intent known to both the compiler and other readers of the code. The `override` identifier indicates that this function is an override of a preexisting virtual inherited from a base class. The `final` identifier marks the virtual function so it cannot be overridden by derived classes.

3.1.3.5 Strongly Typed `enum`s

In C++03, an `enum` exports its enumerators to the surrounding scope, and the type of its enumerators is determined by the compiler based on the values present in the enumeration. C++11 introduces a new kind of strongly typed enumerator, declared using the keywords `enum class`, which scopes its enumerators just like a class or struct scopes its members, and permits the programmer to specify the underlying type.

```
// C++11
enum class Color : std::int8_t { Red, Green, Blue, White, Black };
Color c = Color::Red;
```

3.1.3.6 Standardized Smart Pointers

In C++11, `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr` provide all the facilities we have come to expect from a solid smart pointer facility (much like the Boost library's smart pointer system). `std::unique_ptr` is used when we want to maintain sole "ownership" over the object being pointed to. If we need to maintain multiple pointers to a single object, reference-counted `std::shared_ptr`s should be used. A `std::weak_ptr` acts like a shared pointer, but it does not contribute to the reference count of the pointed-to object. As such, weak pointers are generally used as "back pointers" or in other situations where the pointer "graph" contains cycles.

3.1.3.7 Lambdas

A *lambda* is an anonymous function. It can be used anywhere a function pointer, functor or `std::function` can be used. The term *lambda* is borrowed from functional languages like Lisp and Scheme.

Lambdas allow you to write the implementation of a functor inline, rather than having to declare a named function externally and pass it in. For example:

```
void SomeFunction(const std::vector& v)
{
    auto pos = std::find_if(std::begin(v),
                            std::end(v),
                            [](int n) { return (n % 2 == 1); });
}
```

3.1.3.8 Move Semantics and Rvalue References

Prior to C++11, one of the less-efficient aspects of the C++ language was the way it dealt with copying objects. As an example, consider a function that multiplies each value within a `std::vector` by a fixed multiplier and returns a new vector containing the results.

```
std::vector<float>
MultiplyAllValues(const std::vector<float>& input,
                  float multiplier)
{
    std::vector<float> output(input.size());
    for (std::vector<float>::const_iterator
         it = input.begin();
         it != input.end();
         it++)
    {
        output.push_back(*it * multiplier);
    }
    return output;
}

void Test()
{
    std::vector<float> v;

    // fill v with some values...

    v = MultiplyAllValues(v, 2.0f);

    // use v for something...
}
```

Any seasoned C++ programmer would balk at this implementation, because this code makes at least one if not two copies of the `std::vector` being returned by the function. The first copy happens when we return the local variable `output` to the calling code—this copy will probably be optimized away by the compiler via the *return value optimization*. But the second copy cannot be avoided: It happens when the return value is copied *back* into the vector `v`.

Sometimes copying data is necessary and desirable. But in this (rather contrived) example, the copying is totally unnecessary because the source object (i.e., the vector returned by the function) is a *temporary* object. It will be thrown away immediately after being copied into `v`. Most good C++ programmers (again, prior to C++11) would probably suggest that we rewrite the function as follows to avoid the unnecessary copying:

```
void MultiplyAllValues(std::vector<float>& output,
                       const std::vector<float>& input,
                       float multiplier)
{
    output.resize(0);
    output.reserve(input.size());

    for (std::vector<float>::const_iterator it = input.begin();
         it != input.end();
         it++)
    {
        output.push_back(*it * multiplier);
    }
}
```

Or we might consider making the function less general-purpose by having it modify its input in place.

C++11 provides a mechanism that allows us to rectify these kinds of copying problems *without* having to change the function signature to pass the output object into the function by pointer or reference. This mechanism is known as *move semantics*, and it depends on being able to tell the difference between copying an *lvalue* object and copying an *rvalue* (temporary) object.

In C and C++, an *lvalue* represents an actual storage location in the computer's registers or memory. An *rvalue* is a temporary data object that exists logically but doesn't necessarily occupy any memory. When we write `int a = 7;` the variable `a` is an lvalue, but the literal `7` is an rvalue. You can assign to an lvalue, but you can't assign to an rvalue.

In C++03 and prior, there was no way to handle copying of rvalues differently from copying lvalues. Therefore, the copy constructor and assignment operator had to assume the worst and treat everything like an lvalue. In the case of copying a container object like a `std::vector`, the copy constructor and assignment operator would have to perform a *deep copy*—copying not only the container object itself but all of the data it contains.

In C++11, we can declare a variable to be an *rvalue reference* by using a double ampersand instead of a single ampersand (e.g., `int&& rvalueRef` instead of `int& lvalueRef`). This in turn allows us to write two *overloaded*

variants of both the copy constructor and the assignment operator—one for lvalues and one for rvalues. When we copy an lvalue, we do a full deep copy as always. But when we copy an rvalue (i.e., a temporary object), we needn't perform a deep copy. Instead, we can simply "steal" the contents of the temporary object and *move* them directly into the destination object—hence the term *move semantics.* For example, the copy constructors and assignment operators for a simplified implementation of `std::vector` could be written something like this:

```
namespace std
{
    template<typename T>
    class vector
    {
    private:
        T* m_array;
        int m_count;

    public:

        // lvalue copy ctor
        vector<T>(const vector<T>& original)
            : m_array(nullptr)
            , m_count(original.size())
        {
            if (m_count != 0)
            {
                m_array = new T[m_count];

                if (m_array != nullptr)
                    memcpy(m_array, original.m_array,
                           m_count * sizeof(T));
                else
                    m_count = 0;
            }
        }

        // rvalue "move" ctor
        vector<T>(vector<T>&& original)
            : m_array(original.m_array) // steal the data
            , m_count(original.m_count)
        {
            original.m_array = nullptr; // stolen goods!
            original.m_count = 0;
        }
```

```
        // lvalue assignment operator
        vector<T>& operator=(const vector<T>& original)
        {
            if (this != &original)
            {
                m_array = nullptr;
                m_count = original.size();

                if (m_count != 0)
                {
                    m_array = new T[m_count];

                    if (m_array != nullptr)
                        memcpy(m_array, original.m_array,
                               m_count * sizeof(T));
                    else
                        m_count = 0;
                }
            }
            return *this;
        }

        // rvalue "move" assignment operator
        vector<T>& operator=(vector<T>&& original)
        {
            if (this != &original)
            {
                m_array = original.m_array; // steal the data
                m_count = original.m_count;

                original.m_array = nullptr; // stolen goods!
                original.m_count = 0;
            }
            return *this;
        }

        // ...
    };
}
```

There is one additional subtlety here. An rvalue reference is itself an lvalue (not an rvalue as one might think). In other words, you *can* assign to or modify an rvalue reference variable. That's what allows us to set `original.m_array` to `nullptr` in the example code above. As such, if you want to explicitly invoke a move constructor or move assignment operator on an rvalue reference variable, you have to wrap it in a call to `std::move()` to force the compiler

into thinking your rvalue reference is an rvalue. Confused yet? Never fear, with a bit of practice it will all make sense. For more information on move semantics, see http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html.

3.2 Data, Code and Memory

3.2.1 Numeric Representations

Numbers are at the heart of everything that we do in game engine development (and software development in general). Every software engineer should understand how numbers are represented and stored by a computer. This section will provide you with the basics you'll need throughout the rest of the book.

3.2.1.1 Numeric Bases

People think most naturally in *base ten*, also known as *decimal notation*. In this notation, ten distinct digits are used (0 through 9), and each digit from right to left represents the next highest power of 10. For example, the number $7803 = (7 \times 10^3) + (8 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) = 7000 + 800 + 0 + 3$.

In computer science, mathematical quantities such as integers and real-valued numbers need to be stored in the computer's memory. And as we know, computers store numbers in *binary* format, meaning that only the two digits 0 and 1 are available. We call this a *base-two* representation, because each digit from right to left represents the next highest power of 2. Computer scientists sometimes use a prefix of "0b" to represent binary numbers. For example, the binary number 0b1101 is equivalent to decimal 13, because $0b1101 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13$.

Another common notation popular in computing circles is *hexadecimal*, or *base 16*. In this notation, the 10 digits 0 through 9 and the six letters A through F are used; the letters A through F replace the decimal values 10 through 15, respectively. A prefix of "0x" is used to denote hex numbers in the C and C++ programming languages. This notation is popular because computers generally store data in groups of 8 bits known as *bytes*, and since a single hexadecimal digit represents 4 bits exactly, a *pair* of hex digits represents a byte. For example, the value 0xFF = 0b11111111 = 255 is the largest number that can be stored in 8 bits (1 byte). Each digit in a hexadecimal number, from right to left, represents the next power of 16. So, for example, $0xB052 = (11 \times 16^3) + (0 \times 16^2) + (5 \times 16^1) + (2 \times 16^0) = (11 \times 4096) + (0 \times 256) + (5 \times 16) + (2 \times 1) = 45{,}138$.