C++ for Java Programmers

Timothy A. Budd
Oregon State University
Corvallis, Oregon
USA

December 18, 1998

This is not a blank page.

Preface

The reader envisioned as this book was being developed is a programmer with a year or more experience with Java, who has a good understanding of the language and Java libraries, and who wishes to learn more about the programming language C++. Programs in Java and C++ share a superficial resemblance to each other, but beneath the surface there lie a myriad of practical and philosophical differences. The unwary programmer not cognizant of these differences will encounter a host of problems in moving from one language to another.

This book is not intended to be a complete and thorough introduction to the C++ language. The length of the book alone should be enough to indicate this fact, since most recent introductions to the C++ language run to a thousand pages or more. Instead, this book tries or organize the differences between C++ and Java into a coherent framework that facilitates the transition from one language to the other. Where the reader desires more information on a specific topic, one of the recent descriptions or tutorials on the C++ language should be consulted. Excellent coverage can be found in the book by Stroustrup [Stroustrup 97] or Lippman [Lippman 98].

Further Reading

There are literally hundreds of books on Java and/or C++. I have, of course, seen only a small fraction of these. The following list is therefore quite idiosyncratic, reflecting more than a fair amount of whimsy and chance. These are books that I have read and appreciated.

A good introduction to the C++ language, including the recent changes to the language, can be found in the books by Stroustrup (*The C++ Programming Language* [Stroustrup 97]) or Lippman (*C++ Primer* [Lippman 98]). Slightly less thorough but in some ways more readable descriptions are presented by Eckel (*Using C++* [Eckel 89]), and by Horstmann (*Mastering Object-Oriented Design in C++* [Horstmann 95]).

Two other books authored or co-authored by Bjarne Stroustrup, the designer of C++, present much of the philosophy that lay behind the design of the language (*The Annotated C++ Reference Manual* [Ellis 90], and *The Design and Evolution of C++* [Stroustrup 94]). A collection of papers by others involved in the evolution of the C++ language is provided by Waldo (*The Evolution of C++* [Waldo 93]). Another book by Lippman (*Inside the C++*

Object Model [Lippman 96]) describes the internal C++ view of the world.

The Standard Template Library, a major recent addition to the C++ language, is explained in an earlier book of my own [Budd 98a], as well as in books by Musser [Musser 96] and by Glass [Glass 96].

There are various books that describe good C++ programming style. Perhaps the best of these are the text by Cargill [Cargill 92], and the pair of books by Meyers [Meyers 98, Meyers 96].

A wealth of information in the question and answer style of a FAQ is available in the books that collect FAQ information on C [Summit 96] and C++ [Cline 95]. (There is a Java FAQ book [Kanerva 97], that also has some discussion of C++).

There are several books that are almost the opposite of this book, that is, explanations of Java for programmers familiar with C++. One of the best of these is the book by Chew [Chew 98]. The more recent book by Wigglesworth and Lumby [Wigglesworth 99] covers more of the recent changes to Java. The book by Daconta and others [Daconta 98] is more complete, in that it covers both Java and JavaScript. Pappas [Pappas 96] presents material specific to the Borland C++ system. Boone [Boone 96] tends to dwell more on the programming environment and design differences, and less on the differences in the languages.

In an earlier book I have tried to explain object-oriented programming in a language independent fashion, including examples from both Java and C++ [Budd 97].

A book by Coplien [Coplien 92] presents an interesting discussion of many of the more exotic features of C++, for the adventurous reader who wishes to explore further than most programmers ever wish to go.

Marginal Notes

There are four types of marginal notes used in this book to highlight material of particular important.

DEFINE *Definition* A Definition introduces a term that may be unfamiliar to the programmer if their only background is in the language Java.

RULE *Rule* A Rule provides advice that the reader is strongly encouraged to follow. Like all rules, there may be some times when the advice must be rejected, but rules generally reflect years of painful learning concerning the consequences of not performing some action.

WARNING *Warning* A Warning highlights a potential danger that the programmer should be aware of. Often these reflect subtle issues easily overlooked, or places where Java and C++ constructs have similar appearances but different meanings.

NOTE *Note* A Note simply provides an additional or important bit of information that might easily be overlooked.

Acknowledgements

Several people provided useful advice and suggestions both in the conception of this book and comments on the many early drafts of the manuscript. These include Yechiel Kimchi from The Technion, Israel, Joe Bergin, from Pace University, and students Nandhini GanapathiRaman, Thomas Godin, David Hackenyos, and Hari Narayanan, from Oregon State University.

Chapter 1

Basic Philosophical Differences

Given their historical roots, it is not surprising that programs in Java and C++ have a similar appearance. On the other hand, given the contrasting goals and objectives put forth by the designers of the two languages, it is also not surprising that at a deeper level they are very different. Thus, for the Java programmer to understand C++, they must first comprehend a little bit of its history, philosophy, and background.

1.1 The Language C

To appreciate C++ one has to realize that it descends from an earlier language named C. The language C was conceived and implemented at Bell Labs in the early 1970's by a research scientist named Dennis Ritchie. The primary objective for the language was to assist Ritchie and fellow researcher Brian Kernighan in the creation of an operating system for a small computer they had sitting around their lab. This, in turn, was mainly so that they could develop programs for their own use. They called this operating system Unix, a name intended to contrast with Multics, an operating system running on much larger machines of the period. Other researchers in Bell Labs soon became interested in Unix, and its usage spread. In time Unix, and with it C, became popular in both the research and commercial worlds.

The fact that C and Unix developed in tandem is important, because from the first C was a language designed for systems programming. The language emphasized modern (for the time) control flow constructs and data structures, an economy of expression that would eventually draw equal parts praise and damnation, and a useful ability to access the underlying machine at both a very high level of abstraction, as well as at a primitive and direct level.

C was designed so that even a relatively naive compiler could generate reasonably good assembly language code. Consider, for example, the following typical C idiom for copying one string value into another:

```
while (*p++ = *q++)
    ;
```

The statement is a while loop with no body, only a test for completion. The test portion itself seems not to include booleans (the = operator is assignment, not comparison test, a subtle but vital distinction easily overlooked by the novice programmer), but instead relies on the fact that arithmetic values can be converted into booleans, an integer zero being interpreted as false and anything nonzero as true. The test also relies on assignment being an expression, as well as a side effect producing operation. Assignment will return the updated value of the left hand side, and thus the loop will terminate once a zero value has been assigned. Both the expression from which the assignment draws values and the target expression are determined by pointers, the pointers being dereferenced in both cases to obtain the appropriate memory locations. Finally, the increment operators update the pointer values, advancing them to the next memory locations, at the same time that they yield their current contents. The statement implicitly relies on the convention that strings (in C, arrays of character values) are terminated by a null character, a byte with zero value. Although great havoc will ensue if this condition is not satisfied, no run-time checks are performed to ensure its validity. (For example, there is no limit on how far into memory the pointer q can travel before it is determined that a null character will not be found).

The increment and decrement operators, compound assignment operators, pointer arithmetic, and assignments as values, when combined all meant that even relatively complex statements, such as this one, could be realized by short sequences of assembly language instructions.

Features such as the explicit use of pointers, the variety of bit-level operations, and a simple memory model were ideally suited for getting close to the hardware, and creating programs that ran quickly and required little memory. Compilers for C were soon developed for a variety of machines, and the language became a popular alternative to assembly language for systems programming. (Indeed, it is now common to hear C referred to as a "portable assembly language.")

This is not to say that the language was without controversy. From the first, objections have been raised that the language is too concise for human understanding, and the lack of run-time checks made programs more error-prone than necessary. Programmers in C were forced to accommodate themselves to the fact that the typical error message from a running C program often said little more than "there was an error while your program was in, or near, the computer".

1.2 The Development of C++

NOTE *C++ grew out of an earlier language named C*

The language C++ started out as a collection of macros and library routines for the C language.[1] Only after a period of use did it evolve into a new language of its own. Nevertheless, the newly renamed language C++ was explicitly intended to be more-or-less backward compatible with C. That is, any legal C program should still be a legal C++ program.[2]

This decision, to make C++ an extension of the earlier language rather than a totally new language, was cause for confusion during the early years of the 1980's. While the consequence of this decision provided a valuable service to the industry by introducing object-oriented techniques to the large body of C programmers, many people wrongly assumed that it would be easy to retrain a C programmer to be a C++ programmer. Perhaps almost as easy as adding two characters to their job title. What this presumption ignored was the fact that the way one goes about structuring the solution to a problem in the object-oriented fashion is very different from the way one structures a program in the imperative style of C. Only with time and experience were these differences recognized.

Other ramifications of this decision remain. For instance, the C programming language brought with it a rich collection of libraries, for example the standard I/O library. Since C programs were also C++ programs, C++ had to accommodate these as well, even though they were not particularly well matched to the new object-oriented features of the language. The result is that there are now *two* I/O libraries in common use, the older standard I/O library from the C world and the newer stream I/O system more adapted to C++ (see Chapter 10).

NOTE *Efficiency is held up as a primary goal in C*

The underlying philosophy of C; concise representation, pointer arithmetic, omission of run-time checks, the simple memory model, correspondence of arrays and pointers, the elevation of "uncompromising efficiency" as a virtue above almost all others, were carried over into C++. These made their impact on the language in a multitude of ways, both large and small. The designer of the language has stated that an explicit goal was that the programmer should not have to pay (in space or execution time) for features they do not use. To cite one example, it is for this reason that object-oriented polymorphic function dispatch is provided only if the user explicitly requests it (using the `virtual` keyword), and not otherwise. Similarly the language goes to great lengths to store values on the activation record stack, rather than the heap, because stack-based values are almost always more quickly managed than heap-based values. But this efficiency in execution speed is purchased only at the cost of complications in the language semantics, a topic we will address more fully in Chapter 4.

This attitude that programmers should not have to pay for features they do not use permeates the language. Another example will illustrate this influence. Imagine that an integer variable holding a negative number is right shifted by one location. What bit value should be moved into the topmost position? If we look at the machine level, architectural designers are divided on this issue. On some machines a right shift will move a zero into

[1] *Classes: An Abstract Data Type Facility for the C Language*, Bjarne Stroustrup, *ACM Sigplan Notices*, 1982.

[2] This assertion is true in spirit, but not exactly true in fact. The C++ language does necessarily introduce a few new keywords, and there are a very small number of ways in which the two languages have now diverged. However, the heritage of C++ from C cannot be questioned, and the impact of that heritage is the issue being addressed here.

the most significant bit position, while on other machines the sign bit (which, in the case of a negative number, will be one) is extended. Either case can be simulated by the other using software, by means of a combination of tests and masks.

Given this lack of consensus on the part of hardware systems, one can imagine the following argument being made that would lead to the resolution we see in C++. "The situations where the two interpretations differ is not the common use, in fact it may be rather rare, as it only arises when negative values are shifted. If we adopt either convention as a language definition, then at least on some machines *all* right shifts will be impacted, as every right shift will need to test for the condition. However, it is unnecessarily costly to penalize all right shift operations because of a rare condition. We can get around this problem by asserting that the outcome in this situation is not specified by the language, and thus whatever result is produced by the underlying hardware is correct."

WARNING
The result of some operations in C++ is purposely unspecified

There are a half dozen or so similar cases in the language, where the outcome of an operation is left purposely unspecified, largely in order that whatever instruction is provided by the underlying machine can be used and still be said to satisfy the language specifications. For example, another common situation is the result of dividing or taking the remainder of a negative integer value – the result is either rounded towards zero or towards negative infinity depending upon the platform. Yet another example is the order of execution when a statement includes two autoincrements of the same variable, as in the following:

```
a[i++] = i++; // which increment is done first?
```

Java, on the other hand, provides a precise specification in all of these cases, which means that Java must in some cases correct in software for hardware instructions that do not match the language definition.

WARNING
C++ performs fewer run-time checks than Java

Finally, there is a difference in attitude towards run-time checks between Java and C++. Since the beginning of programming countless debugging hours have been spent tracking down problems that were ultimately discovered to be caused by undefined pointer values, or array subscripts out of range; features that Java (with a different philosophical outlook) would have detected by means of automatically generated run-time checks, but which the C++ language does not generally uncover. Once more the C++ attitude is that if run-time checks are important the programmer should explicitly write them, and if not explicitly called for the language should not impose their cost in situations where they may not be necessary.

The preceding paragraphs should not be construed to imply that Bjarne Stroustrup made wrong decisions in designing C++,[3] rather that it is important to understand the forces motivating those decisions in order to understand the language we have today.

[3]Although some have argued so. Typical is the following quote from Andrew Appel: 'Life is too short to spend time chasing down irreproducible bugs, and money is too valuable to waste on the purchase of flaky software. ... One might say, by way of excuse, "but the language in which I program has the kind of address arithmetic that makes it impossible to know the bounds of an array." Yes, and the man who shot his mother and father threw himself upon the mercy of the court because he was an orphan.'[Appel 97]

1.3 The Legacy Problem

Much of the difficulty in dealing with C++ programs comes from the problem of dealing with *legacy code.* Even when new code is being developed, it will often incorporate features from libraries or systems that can be termed legacy. The problem of legacy code is particularly troublesome in the case of C++ because the language has changed over time, not only evolving from C, but also having new features added over a period of many years.

DEFINE
Legacy code is software written for earlier systems or libraries

The following list describes some of the more common aspects of the problem of legacy code:

- The use of libraries, such as the Standard I/O library, that predate the development of C++.
- The use of the preprocessor to create symbolic defined constants, rather than `const` variables. Tricky use of preprocessor macros to save execution time, rather than in-line function invocations. (The original C language did not have either constant variables nor function inlining).
- The use of simple string functions that manipulate arrays of character values, rather than using the newer `string` data type.
- The use of various different names and implementation techniques for representing boolean values, which predate the addition of the `bool` data type.
- The over use of global variables, which predates the understanding of classes as a better encapsulation technique.
- The use of various different container libraries, which predate the adoption of the Standard Template Library.
- The use of various techniques, such as `void *` pointers, to get around the type system interfering with general purpose containers in code that predates the introduction of templates.

Because of the legacy code problem the C++ programmer must not only learn the current (and hopefully best) practices, but must also be conversant with practices of the past.

1.4 The Language Java

Java was originally envisioned as a language for creating systems to be embedded in consumer products, such as VCRs. It was also developed at a time of increasing processor speeds and decreasing memory costs. In this context, efficiency and the ability to generate compact machine code took a backseat to issues of safety and robustness. James Gosling, the creator of Java, borrowed much of the basic syntax of the C++ language, thereby assuring that programmers with considerable experience in the older language would feel comfortable in the new. But instead of efficiency, the new bywords were simplicity and security.

Even Bjarne Stroustrup, the developer of C++, has stated: "Within C++, there is a much smaller and cleaner language struggling to get out"[4]. Some would argue that this language is Java. Simplicity comes about through the elimination of many of the features of C++. The table in Chapter 12 that describes some of the features in C++ that have no correspondence in Java is just one indication of this. In many cases this comes about by eliminating choices that the C++ programmer has to make; for example all methods in Java are potentially polymorphic, instead of the programmer having to decide which are and which are not. This indeed incurs an overhead, but an acceptable one, and so the language comes down on the opposite side of C++ in the tradeoff between simplicity of language and efficiency of execution.

Java eliminates all the situations that in C++ are explicitly left unspecified by the language definition. These include the meaning of shifts and divisions when negative numbers are manipulated, the size of primitive values, and several others. This is a conscious trade-off on the part of the designers that preserves a consistent behavior on all platforms, paying for this with the expense of possibly hiding in software differences in the underlying hardware.

Run-time checks are another tradeoff, this time between efficiency of execution and safety. The C++ attitude is that if safety (say, detecting the use of uninitialized values) is important the programmer should explicitly code it, and if not the program should not pay (in execution time) for the feature. But experience has shown that far too few programmers will spend the effort to explicitly check for uninitialized values, or that their array index values are in range, or that their pointers do point somewhere, and far too many programming errors result as consequence. Thus, Java comes down on the opposite side of this divide, and will always verify array index bounds, check for the use of undefined variables, and perform other run-time checkable tests.

Another area of philosophical difference is memory management. C++ leaves the management of dynamically allocated memory to the programmer, few of whom will actually perform this task correctly. Java, on the other hand, provides a garbage collection system that will scurry about behind the scenes taking care of memory management tasks for the programmer. This increases execution time, but results in fewer programming errors.

Java also benefited from being developed later, after many years experience with C++. The problems involved in the explicit manipulation of pointer values were by then legendary. An appreciation of the importance of object-oriented features as an improvement over imperative software development was simply not possible when C++ was being designed, but was clear by the middle 1990's. And the designer of Java could draw upon many years of experimentation with libraries and additions to C++, such as with threads packages or with the exception mechanism, both of which have precursors in the C++ world.

A final example of the differences in philosophy between the two languages is the memory model (which is actually distinct from the issue of memory management discussed above).

[4][Stroustrup 94], Section 9.4.4.

The C++ memory model is very close to the underlying machine, and therefore can be very efficiently implemented. But some aspects of this model have very unfortunate consequences for the object-oriented portions of the language, in particular an interaction between memory use and polymorphic method binding. Values that are truly polymorphic cannot be automatic, and values that are automatic cannot be truly polymorphic.[5] Since polymorphism may or may not be important for any particular problem, the C++ language allows both types of values. Java simplifies the language by having only one object format, but this is purchased only at the expense of always using the more costly heap-based memory model.

Many people, both authors and users, have remarked that Java programs in execution are slow.[6] There are many reasons for this, but the most important one is the philosophical differences we have been outlining. A Java program in execution is simply doing more work than is the equivalent C++ program. Garbage collection, multi-thread management, and run-time checks must use some execution time, regardless of how necessary they are. However, recent innovations in Java implementation (ideas such as Just-In-Time compilers) have improved performance dramatically. Whether this rate of improvement can be sustained is uncertain. On the one hand, there is the run-time cost necessitated by Java. On the other hand, the simpler Java language provides more opportunities for improvement by a good optimizing compiler than does C++. How these two forces will balance, and whether Java run-times can ever consistently come close to C++ performance is an open question.

1.5 The Better Language

It is not the intent of this book to argue that one language is "better" in any sense than the other; indeed, such a question is almost meaningless since the objectives and intended purposes of the two languages are so dissimilar. Rather, each language should be appreciated on its own merits, for the way it goes about addressing the problems of particular concern. Understanding Java requires knowing that it was intended for developing programs that would work correctly and securely in systems with minimal interaction (such as embedded systems), even at the cost of execution time. C++ was designed to facilitate the creation of efficient and small executable files for applications such as systems programming. In the end, both languages can be appreciated on their merits as tools to address the problems they were intended to solve, without making any detraction from the other.

1.6 Further Reading

Bjarne Stroustrup has discussed his motivations in designing C++ in several places, notably in two books [Ellis 90, Stroustrup 94], and in journal articles [Stroustrup 98].

[5] This will be explained in more detail in Chapter 4.

[6] See [Tyma 98], for example.

James Gosling discussed the development of Java in an invited talk at the 1996 OOPSLA conference (James Gosling, *The Feel of Java*, unpublished talk), and in the book that currently represents the definitive description of the language [Arnold 98].

Test Your Understanding

1. True or False:
   (a) The language Java is based on an earlier language named J.
   (b) The language C was developed in tandem with the Unix operating system.
   (c) The value 7 can be used as a boolean in C++.
   (d) The language C++ was originally a set of macros called simply Classes.
   (e) Any legal C program is a legal C++ program.
   (f) In C++ efficiency is held as a virtual above all else.
   (g) The exact meaning of some integer division or remainder operations is left unspecified by the C++ language.
   (h) Java was designed as a language for writing controllers to be embedded in consumer products, such as VCRs.
2. Where was the language C original developed?
3. For what purpose was the language C originally developed?
4. What are some of the reasons C became popular as a systems programming language?
5. What are some of the advantages that C++ derived from being an extension of C? What were some of the disadvantages?
6. Why is there an inherent conflict between uncompromising efficiency and portability?
7. Why is there an inherent conflict between uncompromising efficiency and run-time safety?
8. How many different interpretations are possible for the following sequence of statements:

```
i = 4;
a[i++] = i++;
```

9. What are some aspects of the legacy problem?

10. What was the original purpose for the language Java? Contrast this with the original purpose for C++. How are the two different purposes reflected in the languages?

11. For what types of programs is an emphasis on efficiency above all other concerns an appropriate decision? For what types of programs is it not appropriate?

Chapter 2

Fundamental Data Types

We begin with a discussion of the fundamental data types, such as integers, characters and floating point values. In neither Java nor C++ are these values considered to be objects, in the technical sense of the world. Thus, they lack the glamor and allure currently associate with the buzzword "object-oriented", and can easily be thought of as ordinary and pedestrian. Nevertheless, they are the workhorse elements with which all real activity is eventually performed.

There are also a number of surprising differences in the way the two languages handle these basic data types.

2.1 Integers

Both the C++ and Java languages have the notion that integers can be both short and long, in addition to their natural representations. In Java a short integer is explicitly a sixteen bit quantity, an integer a 32 bit quantity, and a long a 64 bit quantity. The C++ standard, on the other hand, is mute concerning the number of bits assigned to each, except to note that an integer value must be at least as large as a short integer, and a long integer must be at least as large as a simple integer. Thus, it would be perfectly legal for a compiler to, for example, use thirty-two bit quantities for all three. It is not uncommon for either a short or a long to be the same size as a simple integer.

WARNING
long and/or short may have the same size as integer

In C++ the designations long and short are modifiers for the integer data type, instead of type names in their own right. Thus it is legal to declare a value as both short and integer. However, it is also possible to use the modifiers by themselves, in which case the base type integer is understood:

```
short int x; // declare x as a small integer
long y; // declare y as long integer
```

The modifier long can also be applied to double precision values (see Section 2.2).

DEFINE *An unsigned integer can only hold nonnegative values*

Another pair of modifiers that can be applied to integer are signed and unsigned. An unsigned value can only hold quantities that are greater than or equal to zero, however typically they can maintain numbers that are larger than those represented by a signed quantity that uses the same number of bits. For example, a sixteen bit signed integer variable can hold values between −32768 and 32767, whereas an unsigned sixteen bit integer can maintain values between zero and 65535.

WARNING *Assigning a negative value to an unsigned variable is confusing*

The language permits a signed value to be assigned to an unsigned variable without casts or warnings. However, if the signed value is negative the result will be an unexpectedly large number:

```
int i = -3;
unsigned int j = i;
cout << j << endl; // will print very large positive integer
```

NOTE *Integer division involving negative numbers is platform dependent*

There is little agreement among machine designers in the exact meaning of integer division when one or both arguments are negative. On some machines the integer division −23/4 will yield −5, the smallest integer greater than the algebraic quotient, while on other machines the same calculation will yield −6, the largest integer less than the algebraic quotient. For this reason the C++ language definition leaves the meaning in this situation undefined, so that language implementors will be free to use the "natural" instruction provided by the underlying hardware. The language Java, on the other hand, explicitly states that integer division truncates towards zero, so that −23/4 will yield −5.

The C++ language definition insists that the following equality must always be preserved:

```
a == (a/b)*b + a%b
```

RULE *Never use the remainder operator with negative values*

Because division involving negative numbers is not completely defined, a similar situation holds with respect to the remainder operator %. In C++ the result of the calculation 21 % −5 is machine dependent, and can either be 1 or −1, matching whatever interpretation is selected for division. In Java it is specified as −1.

Right shifts are also explicitly underdefined in the language. A right shift of an signed quantity can either fill the high order bits with zero values, or extend the sign bit. Both choices are permitted by the language definition, and on any particular machine the alternative elected will probably depend on the interpretation provided by the instruction on the underlying hardware. A right shift of an unsigned quantity must always fill with zero values, and corresponds to the Java >>> operator, which is not part of the C++ language. The effect of either a right or a left shift where the right argument is negative, or where it is larger than the number of bits in the left argument, is undefined.

The modifiers signed and unsigned are orthogonal to long and short, thus permitting a large number of combinations:

```
unsigned long a; // can hold largest integer value
signed short int b;
```

C++ does not recognize the Byte data type in Java, instead the data type signed char is often used to represent byte-sized quantities.

NOTE
signed characters are often used to hold very small integer values

2.1.1 Characters

A character value in C++ is typically only an eight-bit quantity, although again the language definition only provides a minimum length, and a compiler that devoted sixteen bits to each character (as does Java) would in theory be legal. As in Java, it is legal to perform arithmetic on characters, and this is indeed much more common in C++ programs than in Java programs.

The Java Unicode escape format, for example `'\u0ABC'`, is not recognized by C++. However, arbitrary character literals can be represented using their octal value representation. The ASCII character 2, for example, is represented as `'\062'`. Hexadecimal constants can also be written, by beginning with the text 0x, as in `'\0xFF'`.

As a type, characters can be signed or unsigned. An explicitly signed character is typically used more as a very short integer value than as a true character, as literal character values are represented the same whether they are signed or unsigned.

NOTE
Wide characters are a recent addition to the C++ language

A relatively recent addition to the C++ language is the data type `w_char`, a "wide character" that is explicitly larger than a normal character.[1] Typically the name `w_char` is simply an alias for another integer data type, such as short.

2.1.2 Booleans

The boolean data type is named bool in C++, instead of boolean as in Java. This is a relatively recent addition to the C++ language, and is still somewhat infrequently used. Historically, integer values were used to represent boolean quantities. A nonzero arithmetic quantity was interpreted as true, while a zero value was false. It is still legal to use integers in this fashion, for example to control a while loop:

NOTE
Booleans are a recent addition to the C++ language

```
int i = 10;
while (i) { // will loop until i is zero
    ...
    i--;
```

[1] It would perhaps have been more logical to use the name long char to represent sixteen bit characters. One can conjecture that this was not done because the type name is defined as an alias for a short integer by means of a typedef statement, and therefore cannot be represented by a two word name.

```
}
```

Needless to say, while legal this usage is somewhat more error prone than the explicit use of boolean values. For example, should the variable i somehow be negative when the while loop begins, then it will create an infinite loop (or at least a very long one).

One place the integer-as-boolean interpretation is widely used is in the manipulation of string values. The string copy idiom described in the previous chapter is a typical example:

```
while (*p++ = *q++) ;
```

NOTE *Integer and pointer types can be used as boolean values*

Here the program will terminate when a character (that is, an integer) zero value is copied by the assignment. All nonzero values will be interpreted as true, while the zero value is interpreted as false.

The bool data type reveals its heritage by the fact that it is still considered to be an explicit integer data type, although it cannot be signed or unsigned. For example, arithmetic operations can be applied to bool values, which result in the bool value being converted into an integer (zero for false, one for true). Similarly, integer results can be assigned to bool variables, in which case a nonzero value is converted to 1:

```
bool test = true;
int i = 2 + test; // i is now 3
test = test - 1; // test is now 0, or false
```

Even pointer values can be used as booleans, with the interpretation that the value is considered false if it is null, and true otherwise. Thus, it is not uncommon to see pointer variables being tested by an if statement in the following fashion:

```
aClass * aPtr; // declare a pointer variable
...
if (aPtr)    // will be true if aPtr is not null
    ...
```

Note that the test to determine if a pointer is non-null is different from a test to see if the object the pointer references is non-zero, as we saw in the string copy example.

WARNING *Legacy code can contain different boolean abstractions*

Perhaps more than other features, the boolean data abstraction is an area where the programmer can expect to encounter problems with legacy code (see Section 1.3). Because the bool data type is a relatively recent addition to the language, but one that nevertheless has obvious application, there were various competing alternative designs for implementing this data type in the days prior to the C++ standard. For example, some schemes implemented boolean values as simple integers, while other techniques used an enumerated data type. These alternatives differed not only in their implementation, but also in their naming conventions.

To cite just one example, users developing code using the MFC classes on Windows systems will encounter methods that require or return a value of type BOOL. This type is distinct from the bool data type, and care must be taken in mixing the two. Many other schemes are still commonly found in different situations.

2.1.3 Bit Fields

A seldom used feature of C++ allows the programmer to specify explicitly the number of bits to be used in the representation of an integer value. This is often used to pack several different binary values into small structure, such as an eight-bit byte:

```
struct infoByte {
    int on:1; // one-bit value, 0 or 1
    int :4; // four bit padding, not named
    int type: 3; // three bit value, 0 to 7
};
```

The exact layout of the bits is implementation dependent. As a practical matter the use of bit-fields often saves neither time nor space, since in the generated assembly language more complex code is needed to extract or set such fields.

RULE
Don't use bit fields

2.2 Floating Point Values

Floating point quantities are represented in three ranges of magnitude, float, double and long double. The type double is the most commonly used type, for example floating point literals are implicitly defined as double precision. Similarly all mathematical routines in the standard library use double instead of float. In fact, there is almost no reason why any value should ever be declared as float.

RULE
Never use float, use double instead

C++ is much more flexible with conversions than is Java. For example, assigning a floating point value to an integer variable is illegal in Java without a cast, but perfectly acceptable in C++:

```
int i;
double d = 3.14;
i = d; // may generate a warning
```

Better compilers may generate a warning on this statement, but nothing more. When constructors are used as conversion operators, or when explicit conversion operators are present (see Section 7.15), the programmer should take great care as these operations can be invoked implicitly without any indication being given in the program.

WARNING
Math routines will not throw an exception on error
RULE
Always check errno

The standard mathematical library routines (see Section A.6 in Appendix A) will never throw an exception, and will seldom halt execution unless the underlying hardware throws a hardware fault. An integer division by zero will often cause the latter behavior. Instead of halting execution, the standard routines will set a global variable named errno. It is the responsibility of the programmer to check this value after each invocation:

```
double d = sqrt(-1); // should generate error
if (errno == EDOM)
    ... // but only caught if checked
```

Java supports three floating point values that are not numbers, these are Nan, NEGATIVE_INFINITY, and POSITIVE_INFINITY. Such facilities are permitted in a platform dependent fashion in C++, but are not required.

2.3 Enumerated Values

Despite the similar name, an enumerated value has nothing in common with the Enumeration class in Java. An enum declaration in C++ creates a distinct integer type with named constants:

```
enum color {red, orange, yellow};
```

The values red, orange and yellow become named constants after the point of this declaration. A value declared as color can only hold values of this type.

The names of enumerated constants must be distinct. The following could generate an error:

```
enum fruit {apple, pear, orange}; // error: orange redefined
```

Enumerated constants can be converted into integers, and can even have their own internal integer values explicitly specified:

```
enum shape {circle=12, square=3, triangle};
```

The only operation defined for enumerated data types is assignment. An enumerated constant can be assigned to an integer and incremented, but the resulting value must then be cast back into the enumerated data type before it can be assigned to a variable. The validity of the cast is not checked.

```
fruit aFruit = pear;
int i = aFruit; // legal conversion
```

```
i++; // legal increment
aFruit = fruit(i); // fruit is probably now orange
i++;
aFruit = fruit(i); // fruit value is now undefined
```

Cast operations can be written using either the form type(value) or using the older (type)value syntax that is common to both C++ and Java. The latter form is nowdays generally discouraged, although there are situations where it is still applicable. It is not legal to change a pointer type, for example, by writing

NOTE *Cast operations can be written in two different forms*

```
int * i;
char * c;

c = char *(i); // error: not legal syntax
```

However, in this situation a static_cast (Section 6.3) would be even better.

2.4 The void type

As in Java, the data type void is used to represent a method or function that does not yield a result. In C++ the type can also be used as a pointer type, to describe a "universal" pointer that can hold a pointer to any ptype of value. This usage will be discussed in more detail in Chapter 3, when we discuss pointers.

2.5 Arrays

An array in C++ can be created by simply declaring the type and the number of elements. It does not need to be allocated using the new directive, as in Java. When declared in this form, the number of elements must be a value determined at compile time.

NOTE *An array need not be allocated using* new

```
int data[100]; // create an array of 100 elements
```

The number of elements can often be omitted. This is true, for example, if the array has an explicit initialization clause:

```
char text[ ] = "an array of characters";
int limits[ ] = {10, 12, 14, 17, 0};
```

Note that the square brackets follow the name. It is not legal to place the square brackets after the type, as in Java:

```
double[ ] limits = {10, 12, 14, 17, 0}; // legal Java, not C++
```

The limits can also be omitted when arrays are passed as arguments to a function:

```
    // compute average of an array of data values
double average (int n, double data[ ] )
{
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += data[i];
    }
    return sum / n;
}
```

WARNING
In C++ arrays do not know how many elements they contain

Unlike Java, arrays are not objects, and do not possess any methods. They do not "know" their extent. The only operation that is normally performed on an array is the subscript.

There is a close association in C++ between arrays and pointers. We will explore this later in Section 3.6.

2.6 Structures and Unions

Before the advent of object-oriented languages, many programming languages included the concept of a *structure*. A structure is like a class definition that includes only data fields, in which all access is public, and which does not use inheritance. That is, a structure is simply a way of packaging a collection of data fields together as one unit.

The C++ struct data type is heir to this tradition in languages, (in particular, is upward compatible with the earlier C language), but moves slightly in the direction of the more complete class facility. In fact, the major difference in C++ between a struct and a class is that the access to members in structures is by default public, rather than private, as in classes.

```
    // holds an int, a double, AND a pointer
struct myStruct {
    int i;
    double d;
    anObject * p;
};
```

A union is similar to a structure, but the different data fields all share the same location in memory. They can be thought of as being laid one on top of another. Obviously, only one field can therefore be used at any one time. Unions were commonly used in pre-object-oriented days to create a general purpose data area that could hold many different types of values.

```
    // can hold an int, a double, OR a pointer
union myUnion {
    int i;
    double d;
    anObject * p;
};
```

For the most part, unions have been made unnecessary in object-oriented languages by the introduction of polymorphic variables. That is, rather than a structure that will hold three different types of values, a programmer can create a polymorphic variable that can hold values from three different types of subclasses.

2.7 Object Values

Much of the following chapters will be devoted to the differences in the interpretation of classes and objects in Java and C++. However, we begin this discussion with some simple observations. The first is that Java uses *reference semantics* for assignments. This means that a variable assigned from another variable will actually share the same value. This can be seen by creating a class that is nothing more than a simple box:

```
class box {     // Java box
    public int value;
}

box a = new box();
box b;

a.value = 7;        // set variable a
b = a;              // assign b from a
a.value = 12;       // change variable a
System.out.println("a value " + a.value);
System.out.println("b value " + b.value);
```

The result observed will verify that by changing a we have in fact altered the value of b, since they refer to the same object value.

WARNING
Java and C++ use different semantics for assignment

The language C++, on the other hand, normally uses *copy semantics* for assignment. The equivalent program is superficially the same, but with different results:

```
class box {     // C++ box
public:
    int value;
};

box a; // note, explicit allocation not required
box b;

a.value = 7;
b = a;
a.value = 12;
cout << "a value " << a.value << endl;
cout << "b value " << b.value << endl;
```

The output will show that `a` was assigned the value 7 by the assignment statement, and this value was then copied into the variable `b`. Since a copy was made, this value was independent of the value being held by variable `a`. The variable `a` has subsequently been updated, and now holds the value 12, but unlike Java this change has not modified the value held by the variable `b`.

The C++ language does include the concept of a *reference variable*, which is a variable declared as a direct alias. We will have more to say about reference variables in Chapter 3. However, the correspondence is not exact. A reference variable in C++, for example, can never be reassigned to a new value:

```
box a = new box(); // java reference assignment
box b = a;
b = new box();     // reassignment of reference

box a;          // C++ example
box & b = a;     // reference assignment
box c;
b = c;          // error: not permitted to reassign reference
```

2.8 Functions

The C++ language permits the definition of functions that are not members of any class. Such functions are invoked simply by name, without requiring the specification of a receiver:

```
    // define a function for the maximum
    // of two integer values
int max (int i, int j)
{
    if (i < j) return j;
    return i;
}

int x = ...;
int y = ...;
int z = max(x, y);
```

A *prototype* declaration simply declares the name of a function and the argument types, but does not include a function body:

```
    // declare function max defined elsewhere
int max(int, int);
```

Prototypes are necessary in C++ as every function name with its associated parameter-types must be known to the compiler before it can be used in an invocation.

WARNING *In C++ every function name must be known before it can be used*

2.8.1 Order of Argument Evaluation

A subtle difference between Java and C++ concerns the order of argument evaluation. The language Java explicitly states that arguments are evaluated left to right. Consider the following example program:

WARNING *Order of argument evaluation in C++ is undefined*

```
String s = "going, ";
printTest (s, s, s = "gone ");
...
void printTest (String a, String b, String c)
{
    System.out.println(a + b + c);
}
```

The output will always be “going, going, gone” as the first two arguments will be evaluated before the assignment to the third is performed. In C++, on the other hand, the order of argument evaluation is left implementation dependent. Many systems (but not all) will evaluate arguments right to left, not left to right. On these systems the output will be “gone gone gone”.

2.8.2 The Function Main

NOTE *In C++ main is a function outside of any class*

As in Java, execution in C++ programs begins in a function named main. Unlike Java, this function is not part of any class. The function need not, should not, be declared as static. Earlier versions of the C++ language permitted the return type for main to be declared as void, and most compilers will still accept this form. However, the C++ language definition now requires that the return type be declared as int, with the integer value indicating the success or failure of the program. A return value of zero indicates successful execution, while a nonzero value is interpreted as unsuccessful. (What the operating system does with a nonzero return value is platform dependent, and not defined by the language).

RULE *Always return zero on successful completion of the main program*

The function can either be written with no arguments, or with two. When written in the two argument form, the first argument is an integer value and the second an array of pointers to character values (that is, strings):

```
int main (int argc, char *argv[ ])
{
    cout << "executing program " << argv[0] << '\n';
    return 0; // execution successful
}
```

The integer argument is a count on the number of entries in the character array. The relationship between pointers and arrays shown in this example will be discussed in Chapter 3. The manner in which strings are handled in C++ is discussed in Chapter 8.

WARNING *The first command line argument in C++ is always the application name*

The Java programmer will notice one difference in the command line array, in Java the array values consist entirely of the command line arguments, while in C++ the first (that is, zero indexed) element is the executable program name, and the first actual argument is found at index position 1.

2.8.3 Alternative Main Entry Points

Note that main is the entry point for programs that is specified by the language, but individual libraries may provide their own version of main and then require a different entry point. For example, many Windows graphical systems come with their own main routine already written, which will perform certain initialization before invoking a different function (such as WinMain). The MFC (Microsoft Foundation Classes) library takes this one step further, and eliminates the main routine altogether and instead begins execution when an instance of the application class (a subclass built on top of a MFC-provided class) is created.

Test Your Understanding

1. In what ways are the data types short and long different in C++ than in Java?

2. What is the difference between a signed and an unsigned integer value?
3. How is it that integer division can result in different answers on different machines?
4. How many bits does the language specify for a char data type?
5. What value will be assigned to variable i by the following program?

```
signed char a = '2' * 4;
int i = a;
```

6. How many times will the following program loop?

```
int i = 16;
while (i)
    if (! (i % 2)) i += 3;
    else i = i >> 1;
```

7. What will the value of variable i be after the following:

```
int i = 3;
bool j = i;
i = j;
```

8. What is the effect of the following program?

```
bool b = true;
for (int i = 0; i < 10; i++)
    if (b -= 1)
        cout << "yes";
    else
        cout << "no";
```

9. What are the three legal types of floating point values?
10. What are the two varieties of names an enumerated type declaration creates?
11. What are the two different ways a cast operation can be written?
12. Write a declaration for an array of 100 double precision values.
13. Write a declaration for an array of 100 pointers to double precision values.

14. What are some ways that a C++ array is different from a Java array?
15. Explain why the procedure average in Section 2.5 requires two arguments, while a similar procedure in Java could be written with only one argument.
16. In what ways in a structure different from a class? In what ways is the C++ idea of a structure different from the historical concept of the type found in languages such as C?
17. What is the danger in the following program fragment?

```
union {
    int i;
    double d;
    } dataFields;

dataFields.d = 3.14159;
dataFields.i = 7;
double x = 2 * dataFields.d;
```

18. What is the difference between an assignment performed using reference semantics and one that uses copy semantics?
19. What is a function prototype?
20. How is the main function in C++ different from the main function in Java?
21. Explain the relationship between the values printed by a signed negative integer, and the value printed when assigned to an unsigned variable.
22. Write a program that will empirically determine the maximum and minimum values for both the signed and unsigned versions of the integer data types char, short, int and long.
23. Write a program to test the division and remainder operations with negative integers on a particular platform, then write a rule that explains your observed results.
24. Empirically investigate the order of evaluation rules for several different platforms.
25. Write a procedure that takes as argument an array of double precision values, and returns the median value in the array.