4

Types and Declarations

Accept nothing short of perfection!
– anon

Perfection is achieved
only on the point of collapse.
– C. N. Parkinson

Types — fundamental types — Booleans — characters — character literals — integers — integer literals — floating-point types — floating-point literals — sizes — ***void*** — enumerations — declarations — names — scope — initialization — objects — ***typedef***s — advice — exercises.

4.1 Types [dcl.type]

Consider

```
x = y+f(2);
```

For this to make sense in a C++ program, the names ***x***, ***y***, and ***f*** must be suitably declared. That is, the programmer must specify that entities named ***x***, ***y***, and ***f*** exist and that they are of types for which **=** (assignment), **+** (addition), and **( )** (function call), respectively, are meaningful.

Every name (identifier) in a C++ program has a type associated with it. This type determines what operations can be applied to the name (that is, to the entity referred to by the name) and how such operations are interpreted. For example, the declarations

```
float x;        // x is a floating-point variable
int y = 7;      // y is an integer variable with the initial value 7
float f(int);   // f is a function taking an argument of type int and returning a floating-point number
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

would make the example meaningful. Because ***y*** is declared to be an ***int***, it can be assigned to, used in arithmetic expressions, etc. On the other hand, ***f*** is declared to be a function that takes an ***int*** as its argument, so it can be called given a suitable argument.

This chapter presents fundamental types (§4.1.1) and declarations (§4.9). Its examples just demonstrate language features; they are not intended to do anything useful. More extensive and realistic examples are saved for later chapters after more of C++ has been described. This chapter simply provides the most basic elements from which C++ programs are constructed. You must know these elements, plus the terminology and simple syntax that goes with them, in order to complete a real project in C++ and especially to read code written by others. However, a thorough understanding of every detail mentioned in this chapter is not a requirement for understanding the following chapters. Consequently, you may prefer to skim through this chapter, observing the major concepts, and return later as the need for understanding of more details arises.

4.1.1 Fundamental Types [dcl.fundamental]

C++ has a set of fundamental types corresponding to the most common basic storage units of a computer and the most common ways of using them to hold data:

§4.2 A Boolean type (***bool***)
§4.3 Character types (such as ***char***)
§4.4 Integer types (such as ***int***)
§4.5 Floating-point types (such as ***double***)

In addition, a user can define

§4.8 Enumeration types for representing specific sets of values (***enum***)

There also is

§4.7 A type, ***void***, used to signify the absence of information

From these types, we can construct other types:

§5.1 Pointer types (such as ***int****)
§5.2 Array types (such as ***char***[])
§5.5 Reference types (such as ***double***&)
§5.7 Data structures and classes (Chapter 10)

The Boolean, character, and integer types are collectively called *integral types*. The integral and floating-point types are collectively called *arithmetic types*. Enumerations and classes (Chapter 10) are called *user-defined types* because they must be defined by users rather than being available for use without previous declaration, the way fundamental types are. In contrast, other types are called *built-in types.*

The integral and floating-point types are provided in a variety of sizes to give the programmer a choice of the amount of storage consumed, the precision, and the range available for computations (§4.6). The assumption is that a computer provides bytes for holding characters, words for holding and computing integer values, some entity most suitable for floating-point computation, and addresses for referring to those entities. The C++ fundamental types together with pointers and arrays present these machine-level notions to the programmer in a reasonably implementation-independent manner.

For most applications, one could simply use ***bool*** for logical values, ***char*** for characters, ***int*** for integer values, and ***double*** for floating-point values. The remaining fundamental types are

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

variations for optimizations and special needs that are best ignored until such needs arise. They must be known, however, to read old C and C++ code.

4.2 Booleans [dcl.bool]

A Boolean, ***bool***, can have one of the two values ***true*** or ***false***. A Boolean is used to express the results of logical operations. For example:

```
void f(int a, int b)
{
    bool b1 = a==b;      // = is assignment, == is equality
    // ...
}
```

If ***a*** and ***b*** have the same value, ***b1*** becomes ***true***; otherwise, ***b1*** becomes ***false***.

A common use of ***bool*** is as the type of the result of a function that tests some condition (a predicate). For example:

```
bool is_open(File*);

bool greater(int a, int b) { return a>b; }
```

By definition, ***true*** has the value ***1*** when converted to an integer and ***false*** has the value ***0***. Conversely, integers can be implicitly converted to ***bool*** values: nonzero integers convert to ***true*** and ***0*** converts to ***false***. For example:

```
bool b = 7;      // bool(7) is true, so b becomes true
int i = true;    // int(true) is 1, so i becomes 1
```

In arithmetic and logical expressions, ***bool***s are converted to ***int***s; integer arithmetic and logical operations are performed on the converted values. If the result is converted back to ***bool***, a ***0*** is converted to ***false*** and a nonzero value is converted to ***true***.

```
void g()
{
    bool a = true;
    bool b = true;

    bool x = a+b;   // a+b is 2, so x becomes true
    bool y = a|b;   // a|b is 1, so y becomes true
}
```

A pointer can be implicitly converted to a ***bool*** (§C.6.2.5). A nonzero pointer converts to ***true***; zero-valued pointers convert to ***false***.

4.3 Character Types [dcl.char]

A variable of type ***char*** can hold a character of the implementation's character set. For example:

```
char ch = 'a';
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Almost universally, a **char** has 8 bits so that it can hold one of 256 different values. Typically, the character set is a variant of ISO-646, for example ASCII, thus providing the characters appearing on your keyboard. Many problems arise from the fact that this set of characters is only partially standardized (§C.3).

Serious variations occur between character sets supporting different natural languages and also between different character sets supporting the same natural language in different ways. However, here we are interested only in how such differences affect the rules of C++. The larger and more interesting issue of how to program in a multi-lingual, multi-character-set environment is beyond the scope of this book, although it is alluded to in several places (§20.2, §21.7, §C.3.3).

It is safe to assume that the implementation character set includes the decimal digits, the 26 alphabetic characters of English, and some of the basic punctuation characters. It is not safe to assume that there are no more than 127 characters in an 8-bit character set (e.g., some sets provide 255 characters), that there are no more alphabetic characters than English provides (most European languages provide more), that the alphabetic characters are contiguous (EBCDIC leaves a gap between ´i´ and ´j´), or that every character used to write C++ is available (e.g., some national character sets do not provide { } [ ] | \; §C.3.1). Whenever possible, we should avoid making assumptions about the representation of objects. This general rule applies even to characters.

Each character constant has an integer value. For example, the value of ´b´ is **98** in the ASCII character set. Here is a small program that will tell you the integer value of any character you care to input:

```
#include <iostream>

int main()
{
        char c;
        std::cin >> c;
        std::cout << "the value of ´" << c << "´ is " << int(c) << ´\n´;
}
```

The notation **int(c)** gives the integer value for a character **c**. The possibility of converting a **char** to an integer raises the question: is a **char** signed or unsigned? The 256 values represented by an 8-bit byte can be interpreted as the values **0** to **255** or as the values **-127** to **127**. Unfortunately, which choice is made for a plain **char** is implementation-defined (§C.1, §C.3.4). C++ provides two types for which the answer is definite; **signed char**, which can hold at least the values **-127** to **127**, and **unsigned char**, which can hold at least the values **0** to **255**. Fortunately, the difference matters only for values outside the **0** to **127** range, and the most common characters are within that range.

Values outside that range stored in a plain **char** can lead to subtle portability problems. See §C.3.4 if you need to use more than one type of **char** or if you store integers in **char** variables.

A type **wchar_t** is provided to hold characters of a larger character set such as Unicode. It is a distinct type. The size of **wchar_t** is implementation-defined and large enough to hold the largest character set supported by the implementation's locale (see §21.7, §C.3.3). The strange name is a leftover from C. In C, **wchar_t** is a **typedef** (§4.9.7) rather than a built-in type. The suffix **_t** was added to distinguish standard **typedef**s.

Note that the character types are integral types (§4.1.1) so that arithmetic and logical operations (§6.2) apply.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

4.3.1 Character Literals [dcl.char.lit]

A character literal, often called a character constant, is a character enclosed in single quotes, for example, ´*a*´ and ´*0*´. The type of a character literal is ***char***. Such character literals are really symbolic constants for the integer value of the characters in the character set of the machine on which the C++ program is to run. For example, if you are running on a machine using the ASCII character set, the value of ´*0*´ is *48*. The use of character literals rather than decimal notation makes programs more portable. A few characters also have standard names that use the backslash \ as an escape character. For example, ***n*** is a newline and ***t*** is a horizontal tab. See §C.3.2 for details about escape characters.

Wide character literals are of the form ***L´ab´***, where the number of characters between the quotes and their meanings is implementation-defined to match the ***wchar_t*** type. A wide character literal has type ***wchar_t***.

4.4 Integer Types [dcl.int]

Like ***char***, each integer type comes in three forms: ''plain'' ***int***, ***signed int***, and ***unsigned int***. In addition, integers come in three sizes: ***short int***, ''plain'' ***int***, and ***long int***. A ***long int*** can be referred to as plain ***long***. Similarly, ***short*** is a synonym for ***short int***, ***unsigned*** for ***unsigned int***, and ***signed*** for ***signed int***.

The ***unsigned*** integer types are ideal for uses that treat storage as a bit array. Using an ***unsigned*** instead of an ***int*** to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables ***unsigned*** will typically be defeated by the implicit conversion rules (§C.6.1, §C.6.2.1).

Unlike plain ***char***s, plain ***int***s are always signed. The signed ***int*** types are simply more explicit synonyms for their plain ***int*** counterparts.

4.4.1 Integer Literals [dcl.int.lit]

Integer literals come in four guises: decimal, octal, hexadecimal, and character literals. Decimal literals are the most commonly used and look as you would expect them to:

```
0   1234   976   12345678901234567890
```

The compiler ought to warn about literals that are too long to represent.

A literal starting with zero followed by ***x*** (***0x***) is a hexadecimal (base 16) number. A literal starting with zero followed by a digit is an octal (base 8) number. For example:

```
decimal:      0      2      63      83
octal:        00     02     077     0123
hexadecimal:  0x0    0x2    0x3f    0x53
```

The letters ***a***, ***b***, ***c***, ***d***, ***e***, and ***f***, or their uppercase equivalents, are used to represent *10*, *11*, *12*, *13*, *14*, and *15*, respectively. Octal and hexadecimal notations are most useful for expressing bit patterns. Using these notations to express genuine numbers can lead to surprises. For example, on a machine on which an ***int*** is represented as a two's complement 16-bit integer, ***0xffff*** is the negative decimal number ***-1***. Had more bits been used to represent an integer, it would have been *65535*.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The suffix ***U*** can be used to write explicitly ***unsigned*** literals. Similarly, the suffix ***L*** can be used to write explicitly ***long*** literals. For example, ***3*** is an ***int***, 3U is an ***unsigned int***, and ***3L*** is a ***long int***. If no suffix is provided, the compiler gives an integer literal a suitable type based on its value and the implementation's integer sizes (§C.4).

It is a good idea to limit the use of nonobvious constants to a few well-commented ***const*** (§5.4) or enumerator (§4.8) initializers.

4.5 Floating-Point Types [dcl.float]

The floating-point types represent floating-point numbers. Like integers, floating-point types come in three sizes: ***float*** (single-precision), ***double*** (double-precision), and ***long double*** (extended-precision).

The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don't have that understanding, get advice, take the time to learn, or use ***double*** and hope for the best.

4.5.1 Floating-Point Literals [dcl.fp.lit]

By default, a floating-point literal is of type ***double***. Again, a compiler ought to warn about floating-point literals that are too large to be represented. Here are some floating-point literals:

```
1.23   .23   0.23   1.   1.0   1.2e10   1.23e-15
```

Note that a space cannot occur in the middle of a floating-point literal. For example, `65.43 e-21` is not a floating-point literal but rather four separate lexical tokens (causing a syntax error):

```
65.43   e   -   21
```

If you want a floating-point literal of type ***float***, you can define one using the suffix ***f*** or ***F***:

```
3.14159265f   2.0f   2.997925F
```

4.6 Sizes [dcl.size]

Some of the aspects of C++'s fundamental types, such as the size of an ***int***, are implementation-defined (§C.2). I point out these dependencies and often recommend avoiding them or taking steps to minimize their impact. Why should you bother? People who program on a variety of systems or use a variety of compilers care a lot because if they don't, they are forced to waste time finding and fixing obscure bugs. People who claim they don't care about portability usually do so because they use only a single system and feel they can afford the attitude that ''the language is what my compiler implements.'' This is a narrow and shortsighted view. If your program is a success, it is likely to be ported, so someone will have to find and fix problems related to implementation-dependent features. In addition, programs often need to be compiled with other compilers for the same system, and even a future release of your favorite compiler may do some things differently from the current one. It is far easier to know and limit the impact of implementation dependencies

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

when a program is written than to try to untangle the mess afterwards.

It is relatively easy to limit the impact of implementation-dependent language features. Limiting the impact of system-dependent library facilities is far harder. Using standard library facilities wherever feasible is one approach.

The reason for providing more than one integer type, more than one unsigned type, and more than one floating-point type is to allow the programmer to take advantage of hardware characteristics. On many machines, there are significant differences in memory requirements, memory access times, and computation speed between the different varieties of fundamental types. If you know a machine, it is usually easy to choose, for example, the appropriate integer type for a particular variable. Writing truly portable low-level code is harder.

Sizes of C++ objects are expressed in terms of multiples of the size of a ***char***, so by definition the size of a ***char*** is ***1***. The size of an object or type can be obtained using the ***sizeof*** operator (§6.2). This is what is guaranteed about sizes of fundamental types:

$$1 \equiv sizeof(char) \le sizeof(short) \le sizeof(int) \le sizeof(long)$$

$$1 \le sizeof(bool) \le sizeof(long)$$

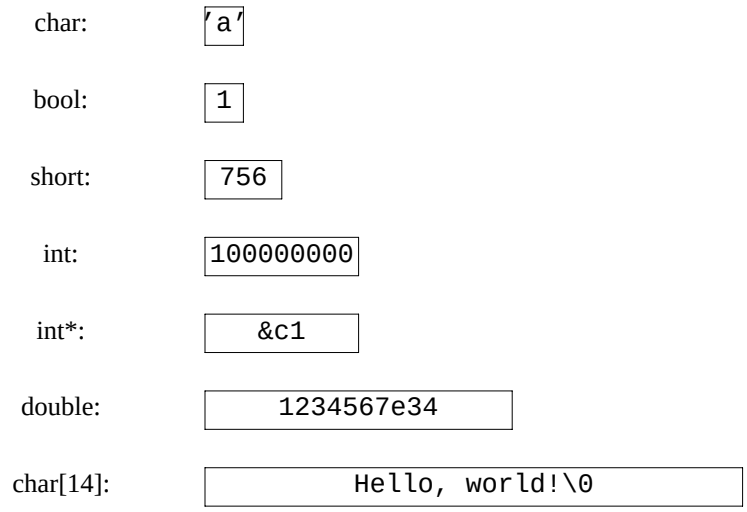
$$sizeof(char) \le sizeof(wchar\_t) \le sizeof(long)$$

$$sizeof(float) \le sizeof(double) \le sizeof(long\ double)$$

$$sizeof(N) \equiv sizeof(signed\ N) \equiv sizeof(unsigned\ N)$$

where ***N*** can be ***char***, ***short int***, ***int***, or ***long int***. In addition, it is guaranteed that a ***char*** has at least 8 bits, a ***short*** at least 16 bits, and a ***long*** at least 32 bits. A ***char*** can hold a character of the machine's character set.

Here is a graphical representation of a plausible set of fundamental types and a sample string:



On the same scale (.2 inch to a byte), a megabyte of memory would stretch about three miles (five km) to the right.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The ***char*** type is supposed to be chosen by the implementation to be the most suitable type for holding and manipulating characters on a given computer; it is typically an 8-bit byte. Similarly, the ***int*** type is supposed to be chosen to be the most suitable for holding and manipulating integers on a given computer; it is typically a 4-byte (32-bit) word. It is unwise to assume more. For example, there are machines with 32 bit ***char***s.

When needed, implementation-dependent aspects about an implementation can be found in **<*limits*>** (§22.2). For example:

```
#include <limits>

int main()
{
        cout << "largest float == " << numeric_limits<float>::max()
             << ", char is signed == " << numeric_limits<char>::is_signed << '\n';
}
```

The fundamental types can be mixed freely in assignments and expressions. Wherever possible, values are converted so as not to lose information (§C.6).

If a value ***v*** can be represented exactly in a variable of type ***T***, a conversion of ***v*** to ***T*** is value-preserving and no problem. The cases where conversions are not value-preserving are best avoided (§C.6.2.6).

You need to understand implicit conversion in some detail in order to complete a major project and especially to understand real code written by others. However, such understanding is not required to read the following chapters.

4.7 Void [dcl.void]

The type ***void*** is syntactically a fundamental type. It can, however, be used only as part of a more complicated type; there are no objects of type ***void***. It is used either to specify that a function does not return a value or as the base type for pointers to objects of unknown type. For example:

```
void x;         // error: there are no void objects
void f();       // function f does not return a value (§7.3)
void* pv;       // pointer to object of unknown type (§5.6)
```

When declaring a function, you must specify the type of the value returned. Logically, you would expect to be able to indicate that a function didn't return a value by omitting the return type. However, that would make the grammar (Appendix A) less regular and clash with C usage. Consequently, ***void*** is used as a ''pseudo return type'' to indicate that a function doesn't return a value.

4.8 Enumerations [dcl.enum]

An *enumeration* is a type that can hold a set of values specified by the user. Once defined, an enumeration is used very much like an integer type.

Named integer constants can be defined as members of an enumeration. For example,

```
enum { ASM, AUTO, BREAK };
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

defines three integer constants, called enumerators, and assigns values to them. By default, enumerator values are assigned increasing from *0*, so *ASM==0*, *AUTO==1*, and *BREAK==2*. An enumeration can be named. For example:

```
enum keyword { ASM, AUTO, BREAK };
```

Each enumeration is a distinct type. The type of an enumerator is its enumeration. For example, *AUTO* is of type *keyword*.

Declaring a variable *keyword* instead of plain *int* can give both the user and the compiler a hint as to the intended use. For example:

```
void f(keyword key)
{
    switch (key) {
    case ASM:
        // do something
        break;
    case BREAK:
        // do something
        break;
    }
}
```

A compiler can issue a warning because only two out of three *keyword* values are handled.

An enumerator can be initialized by a *constant-expression* (§C.5) of integral type (§4.1.1). The range of an enumeration holds all the enumeration's enumerator values rounded up to the nearest larger binary power minus *1*. The range goes down to *0* if the smallest enumerator is non-negative and to the nearest lesser negative binary power if the smallest enumerator is negative. This defines the smallest bit-field capable of holding the enumerator values. For example:

```
enum e1 { dark, light };                    // range 0:1
enum e2 { a = 3, b = 9 };                   // range 0:15
enum e3 { min = -10, max = 1000000 };       // range -1048576:1048575
```

A value of integral type may be explicitly converted to an enumeration type. The result of such a conversion is undefined unless the value is within the range of the enumeration. For example:

```
enum flag { x=1, y=2, z=4, e=8 };  // range 0:15

flag f1 = 5;          // type error: 5 is not of type flag
flag f2 = flag(5);    // ok: flag(5) is of type flag and within the range of flag

flag f3 = flag(z|e);  // ok: flag(12) is of type flag and within the range of flag
flag f4 = flag(99);   // undefined: 99 is not within the range of flag
```

The last assignment shows why there is no implicit conversion from an integer to an enumeration; most integer values do not have a representation in a particular enumeration.

The notion of a range of values for an enumeration differs from the enumeration notion in the Pascal family of languages. However, bit-manipulation examples that require values outside the set of enumerators to be well-defined have a long history in C and C++.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The *sizeof* an enumeration is the *sizeof* some integral type that can hold its range and not larger than *sizeof*(*int*), unless an enumerator cannot be represented as an *int* or as an *unsigned int*. For example, *sizeof*(*e1*) could be *1* or maybe *4* but not *8* on a machine where *sizeof*(*int*)==*4*.

By default, enumerations are converted to integers for arithmetic operations (§6.2). An enumeration is a user-defined type, so users can define their own operations, such as ++ and << for an enumeration (§11.2.3).

4.9 Declarations [dcl.dcl]

Before a name (identifier) can be used in a C++ program, it must be declared. That is, its type must be specified to inform the compiler to what kind of entity the name refers. Here are some examples illustrating the diversity of declarations:

```
char ch;
string s;
int count = 1;
const double pi = 3.1415926535897932385;
extern int error_number;

char* name = "Njal";
char* season[] = { "spring", "summer", "fall", "winter" };

struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<class T> T abs(T a) { return a<0 ? -a : a; }

typedef complex<short> Point;
struct User;
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

As can be seen from these examples, a declaration can do more than simply associate a type with a name. Most of these *declarations* are also *definitions*; that is, they also define an entity for the name to which they refer. For ***ch***, that entity is the appropriate amount of memory to be used as a variable – that memory will be allocated. For ***day***, it is the specified function. For the constant ***pi***, it is the value *3.1415926535897932385*. For ***Date***, that entity is a new type. For ***Point***, it is the type ***complex<short>*** so that ***Point*** becomes a synonym for ***complex<short>***. Of the declarations above, only

```
double sqrt(double);
extern int error_number;
struct User;
```

are not also definitions; that is, the entity they refer to must be defined elsewhere. The code (body) for the function ***sqrt*** must be specified by some other declaration, the memory for the ***int*** variable ***error_number*** must be allocated by some other declaration of ***error_number***, and some other declaration of the type ***User*** must define what that type looks like. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
double sqrt(double d) { /* ... */ }
int error_number = 1;

struct User { /* ... */ };
```

There must always be exactly one definition for each name in a C++ program (for the effects of *#include*, see §9.2.3). However, there can be many declarations. All declarations of an entity must agree on the type of the entity referred to. So, this fragment has two errors:

```
int count;
int count; // error: redefinition

extern int error_number;
extern short error_number;      // error: type mismatch
```

and this has none (for the use of *extern* see §9.2):

```
extern int error_number;
extern int error_number;
```

Some definitions specify a ''value'' for the entities they define. For example:

```
struct Date { int d, m, y; };
typedef complex<short> Point;
int day(Date* p) { return p->d; }
const double pi = 3.1415926535897932385;
```

For types, templates, functions, and constants, the ''value'' is permanent. For nonconstant data types, the initial value may be changed later. For example:

```
void f()
{
    int count = 1;
    char* name = "Bjarne";
    // ...
    count = 2;
    name = "Marian";
}
```

Of the definitions, only

```
char ch;
string s;
```

do not specify values. See §4.9.5 and §10.4.2 for explanations of how and when a variable is assigned a default value. Any declaration that specifies a value is a definition.

4.9.1 The Structure of a Declaration [dcl.parts]

A declaration consists of four parts: an optional ''specifier,'' a base type, a declarator, and an optional initializer. Except for function and namespace definitions, a declaration is terminated by a semicolon. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
char* kings[] = { "Antigonus", "Seleucus", "Ptolemy" };
```

Here, the base type is ***char***, the declarator is ****kings[]***, and the initializer is ***={...}***.

A specifier is an initial keyword, such as ***virtual*** (§2.5.5, §12.2.6) and ***extern*** (§9.2), that specifies some non-type attribute of what is being declared.

A declarator is composed of a name and optionally some declarator operators. The most common declarator operators are (§A.7.1):

*	*pointer*	*prefix*
*const	*constant pointer*	*prefix*
&	*reference*	*prefix*
[]	*array*	*postfix*
()	*function*	*postfix*

Their use would be simple if they were all either prefix or postfix. However, *, [], and () were designed to mirror their use in expressions (§6.2). Thus, * is prefix and [] and () are postfix. The postfix declarator operators bind tighter than the prefix ones. Consequently, ****kings[]*** is a vector of pointers to something, and we have to use parentheses to express types such as ''pointer to function;'' see examples in §5.1. For full details, see the grammar in Appendix A.

Note that the type cannot be left out of a declaration. For example:

```
const c = 7;      // error: no type
gt(int a, int b) { return (a>b) ? a : b; } // error: no return type

unsigned ui;      // ok: 'unsigned' is the type 'unsigned int'
long li;          // ok: 'long' is the type 'long int'
```

In this, standard C++ differs from earlier versions of C and C++ that allowed the first two examples by considering ***int*** to be the type when none were specified (§B.2). This ''implicit ***int***'' rule was a source of subtle errors and confusion.

4.9.2 Declaring Multiple Names [dcl.multi]

It is possible to declare several names in a single declaration. The declaration simply contains a list of comma-separated declarators. For example, we can declare two integers like this:

```
int x, y;         // int x; int y;
```

Note that operators apply to individual names only – and not to any subsequent names in the same declaration. For example:

```
int* p, y;        // int* p;   int y; NOT int* y;
int x, *q;        // int x;    int* q;
int v[10], *pv;   // int v[10];      int* pv;
```

Such constructs make a program less readable and should be avoided.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

4.9.3 Names [dcl.name]

A name (identifier) consists of a sequence of letters and digits. The first character must be a letter. The underscore character _ is considered a letter. C++ imposes no limit on the number of characters in a name. However, some parts of an implementation are not under the control of the compiler writer (in particular, the linker), and those parts, unfortunately, sometimes do impose limits. Some run-time environments also make it necessary to extend or restrict the set of characters accepted in an identifier. Extensions (e.g., allowing the character $ in a name) yield nonportable programs. A C++ keyword (Appendix A), such as ***new*** and ***int***, cannot be used as a name of a user-defined entity. Examples of names are:

```
hello          this_is_a_most_unusually_long_name
DEFINED        foO        bAr          u_name       HorseSense
var0           var1       CLASS        _class       ___
```

Examples of character sequences that cannot be used as identifiers are:

```
012            a fool     $sys         class        3var
pay.due        foo~bar    .name        if
```

Names starting with an underscore are reserved for special facilities in the implementation and the run-time environment, so such names should not be used in application programs.

When reading a program, the compiler always looks for the longest string of characters that could make up a name. Hence, ***var10*** is a single name, not the name ***var*** followed by the number ***10***. Also, ***elseif*** is a single name, not the keyword ***else*** followed by the keyword ***if***.

Uppercase and lowercase letters are distinct, so ***Count*** and ***count*** are different names, but it is unwise to choose names that differ only by capitalization. In general, it is best to avoid names that differ only in subtle ways. For example, the uppercase o (***O***) and zero (***0***) can be hard to tell apart, as can the lowercase L (***l***) and one (***1***). Consequently, ***l0***, ***lO***, ***l1***, and ***ll*** are poor choices for identifier names.

Names from a large scope ought to have relatively long and reasonably obvious names, such as ***vector***, ***Window_with_border***, and ***Department_number***. However, code is clearer if names used only in a small scope have short, conventional names such as ***x***, ***i***, and ***p***. Classes (Chapter 10) and namespaces (§8.2) can be used to keep scopes small. It is often useful to keep frequently used names relatively short and reserve really long names for infrequently used entities. Choose names to reflect the meaning of an entity rather than its implementation. For example, ***phone_book*** is better than ***number_list*** even if the phone numbers happen to be stored in a ***list*** (§3.7). Choosing good names is an art.

Try to maintain a consistent naming style. For example, capitalize nonstandard library user-defined types and start nontypes with a lowercase letter (for example, ***Shape*** and ***current_token***). Also, use all capitals for macros (if you must use macros; for example, ***HACK***) and use underscores to separate words in an identifier. However, consistency is hard to achieve because programs are typically composed of fragments from different sources and several different reasonable styles are in use. Be consistent in your use of abbreviations and acronyms.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

4.9.4 Scope [dcl.scope]

A declaration introduces a name into a scope; that is, a name can be used only in a specific part of the program text. For a name declared in a function (often called a *local name*), that scope extends from its point of declaration to the end of the block in which its declaration occurs. A *block* is a section of code delimited by a { } pair.

A name is called *global* if it is defined outside any function, class (Chapter 10), or namespace (§8.2). The scope of a global name extends from the point of declaration to the end of the file in which its declaration occurs. A declaration of a name in a block can hide a declaration in an enclosing block or a global name. That is, a name can be redefined to refer to a different entity within a block. After exit from the block, the name resumes its previous meaning. For example:

```
int x;                  // global x

void f()
{
    int x;              // local x hides global x
    x = 1;              // assign to local x

    {
        int x;          // hides first local x
        x = 2;          // assign to second local x
    }

    x = 3;              // assign to first local x
}

int* p = &x;            // take address of global x
```

Hiding names is unavoidable when writing large programs. However, a human reader can easily fail to notice that a name has been hidden. Because such errors are relatively rare, they can be very difficult to find. Consequently, name hiding should be minimized. Using names such as ***i*** and ***x*** for global variables or for local variables in a large function is asking for trouble.

A hidden global name can be referred to using the scope resolution operator ::. For example:

```
int x;

void f2()
{
    int x = 1;  // hide global x
    ::x = 2;    // assign to global x
    x = 2;      // assign to local x
    // ...
}
```

There is no way to use a hidden local name.

The scope of a name starts at its point of declaration; that is, after the complete declarator and before the initializer. This implies that a name can be used even to specify its own initial value. For example:

```
int x;
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void f3()
{
    int x = x; // perverse: initialize x with its own (uninitialized) value
}
```

This is not illegal, just silly. A good compiler will warn if a variable is used before it has been set (see also §5.9[9]).

It is possible to use a single name to refer to two different objects in a block without using the `::` operator. For example:

```
int x = 11;

void f4()           // perverse:
{
    int y = x;      // use global x: y = 11
    int x = 22;
    y = x;          // use local x: y = 22
}
```

Function argument names are considered declared in the outermost block of a function, so

```
void f5(int x)
{
    int x;      // error
}
```

is an error because `x` is defined twice in the same scope. Having this be an error allows a not uncommon, subtle mistake to be caught.

4.9.5 Initialization [dcl.init]

If an initializer is specified for an object, that initializer determines the initial value of an object. If no initializer is specified, a global (§4.9.4), namespace (§8.2), or local static object (§7.1.2, §10.2.4) (collectively called *static objects*) is initialized to `0` of the appropriate type. For example:

```
int a;          // means ''int a = 0;''
double d;       // means ''double d = 0.0;''
```

Local variables (sometimes called *automatic objects*) and objects created on the free store (sometimes called *dynamic objects* or *heap objects*) are not initialized by default. For example:

```
void f()
{
    int x;      // x does not have a well-defined value
    // ...
}
```

Members of arrays and structures are default initialized or not depending on whether the array or structure is static. User-defined types may have default initialization defined (§10.4.2).

More complicated objects require more than one value as an initializer. This is handled by initializer lists delimited by `{` and `}` for C-style initialization of arrays (§5.2.1) and structures (§5.7). For user-defined types with constructors, function-style argument lists are used (§2.5.2, §10.2.3).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Note that an empty pair of parentheses `()` in a declaration always means ''function'' (§7.1). For example:

```
int a[] = { 1, 2 };        // array initializer
Point z(1,2);              // function-style initializer (initialization by constructor)
int f();                   // function declaration
```

4.9.6 Objects and Lvalues [dcl.objects]

We can allocate and use ''variables'' that do not have names, and it is possible to assign to strange-looking expressions (e.g., `*p[a+10]=7`). Consequently, there is a need for a name for ''something in memory.'' This is the simplest and most fundamental notion of an object. That is, an *object* is a contiguous region of storage; an *lvalue* is an expression that refers to an object. The word *lvalue* was originally coined to mean ''something that can be on the left-hand side of an assignment.'' However, not every lvalue may be used on the left-hand side of an assignment; an lvalue can refer to a constant (§5.5). An lvalue that has not been declared ***const*** is often called a *modifiable lvalue*. This simple and low-level notion of an object should not be confused with the notions of class object and object of polymorphic type (§15.4.3).

Unless the programmer specifies otherwise (§7.1.2, §10.4.8), an object declared in a function is created when its definition is encountered and destroyed when its name goes out of scope (§10.4.4). Such objects are called automatic objects. Objects declared in global or namespace scope and ***static***s declared in functions or classes are created and initialized once (only) and ''live'' until the program terminates (§10.4.9). Such objects are called static objects. Array elements and nonstatic structure or class members have their lifetimes determined by the object of which they are part.

Using the ***new*** and ***delete*** operators, you can create objects whose lifetimes are controlled directly (§6.2.6).

4.9.7 Typedef [dcl.typedef]

A declaration prefixed by the keyword ***typedef*** declares a new name for the type rather than a new variable of the given type. For example:

```
typedef char* Pchar;
Pchar p1, p2;        // p1 and p2 are char*s
char* p3 = p1;
```

A name defined like this, usually called a '' ***typedef***,'' can be a convenient shorthand for a type with an unwieldy name. For example, ***unsigned char*** is too long for really frequent use, so we could define a synonym, ***uchar***:

```
typedef unsigned char uchar;
```

Another use of a ***typedef*** is to limit the direct reference to a type to one place. For example:

```
typedef int int32;
typedef short int16;
```

If we now use ***int32*** wherever we need a potentially large integer, we can port our program to a machine on which ***sizeof(int)*** is *2* by redefining the single occurrence of ***int*** in our code:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
typedef long int32;
```

For good and bad, ***typedef***s are synonyms for other types rather than distinct types. Consequently, ***typedef***s mix freely with the types for which they are synonyms. People who would like to have distinct types with identical semantics or identical representation should look at enumerations (§4.8) or classes (Chapter 10).

4.10 Advice [dcl.advice]

[1] Keep scopes small; §4.9.4.
[2] Don't use the same name in both a scope and an enclosing scope; §4.9.4.
[3] Declare one name (only) per declaration; §4.9.2.
[4] Keep common and local names short, and keep uncommon and nonlocal names longer; §4.9.3.
[5] Avoid similar-looking names; §4.9.3.
[6] Maintain a consistent naming style; §4.9.3.
[7] Choose names carefully to reflect meaning rather than implementation; §4.9.3.
[8] Use a ***typedef*** to define a meaningful name for a built-in type in cases in which the built-in type used to represent a value might change; §4.9.7.
[9] Use ***typedef***s to define synonyms for types; use enumerations and classes to define new types; §4.9.7.
[10] Remember that every declaration must specify a type (there is no ''implicit ***int***''); §4.9.1.
[11] Avoid unnecessary assumptions about the numeric value of characters; §4.3.1, §C.6.2.1.
[12] Avoid unnecessary assumptions about the size of integers; §4.6.
[13] Avoid unnecessary assumptions about the range of floating-point types; §4.6.
[14] Prefer a plain ***int*** over a ***short int*** or a ***long int***; §4.6.
[15] Prefer a ***double*** over a ***float*** or a ***long double***; §4.5.
[16] Prefer plain ***char*** over ***signed char*** and ***unsigned char***; §C.3.4.
[17] Avoid making unnecessary assumptions about the sizes of objects; §4.6.
[18] Avoid unsigned arithmetic; §4.4.
[19] View ***signed*** to ***unsigned*** and ***unsigned*** to ***signed*** conversions with suspicion; §C.6.2.6.
[20] View floating-point to integer conversions with suspicion; §C.6.2.6.
[21] View conversions to a smaller type, such as ***int*** to ***char***, with suspicion; §C.6.2.6.

4.11 Exercises [dcl.exercises]

1. (∗2) Get the ''Hello, world!'' program (§3.2) to run. If that program doesn't compile as written, look at §B.3.1.
2. (∗1) For each declaration in §4.9, do the following: If the declaration is not a definition, write a definition for it. If the declaration is a definition, write a declaration for it that is not also a definition.
3. (∗1.5) Write a program that prints the sizes of the fundamental types, a few pointer types, and a few enumerations of your choice. Use the ***sizeof*** operator.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

4. (∗1.5) Write a program that prints out the letters ´*a*´..´*z*´ and the digits ´*0*´..´*9*´ and their integer values. Do the same for other printable characters. Do the same again but use hexadecimal notation.
5. (∗2) What, on your system, are the largest and the smallest values of the following types: ***char***, ***short***, ***int***, ***long***, ***float***, ***double***, ***long double***, and ***unsigned***.
6. (∗1) What is the longest local name you can use in a C++ program on your system? What is the longest external name you can use in a C++ program on your system? Are there any restrictions on the characters you can use in a name?
7. (∗2) Draw a graph of the integer and fundamental types where a type points to another type if all values of the first can be represented as values of the second on every standards-conforming implementation. Draw the same graph for the types on your favorite implementation.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.