Appendix C

Technicalities

Deep in the fundamental
heart of mind and Universe,
there is a reason.
– Slartibartfast

What the standard promises — character sets — integer literals — constant expressions — promotions and conversions — multidimensional arrays — fields and unions — memory management — garbage collection — namespaces — access control — pointers to data members — templates — ***static*** members — ***friends*** — templates as template parameters — template argument deduction — ***typename*** and ***template*** qualification — instantiation — name binding — templates and namespaces — explicit instantiation — advice.

C.1 Introduction and Overview

This chapter presents technical details and examples that do not fit neatly into my presentation of the main C++ language features and their uses. The details presented here can be important when you are writing a program and essential when reading code written using them. However, I consider them technical details that should not be allowed to distract from the student's primary task of learning to use C++ well or the programmer's primary task of expressing ideas as clearly and as directly as possible in C++.

C.2 The Standard

Contrary to common belief, strictly adhering to the C++ language and library standard doesn't guarantee good code or even portable code. The standard doesn't say whether a piece of code is good

or bad; it simply says what a programmer can and cannot rely on from an implementation. One can write perfectly awful standard-conforming programs, and most real-world programs rely on features not covered by the standard.

Many important things are deemed *implementation-defined* by the standard. This means that each implementation must provide a specific, well-defined behavior for a construct and that behavior must be documented. For example:

```
unsigned char c1 = 64;      // well-defined: a char has at least 8 bits and can always hold 64
unsigned char c2 = 1256;    // implementation-defined: truncation if a char has only 8 bits
```

The initialization of *c1* is well-defined because a **char** must be at least 8 bits. However, the behavior of the initialization of *c2* is implementation-defined because the number of bits in a **char** is implementation-defined. If the **char** has only 8 bits, the value *1256* will be truncated to *232* (§C.6.2.1). Most implementation-defined features relate to differences in the hardware used to run a program.

When writing real-world programs, it is usually necessary to rely on implementation-defined behavior. Such behavior is the price we pay for the ability to operate effectively on a large range of systems. For example, the language would have been much simpler if all characters had been 8 bits and all integers 32 bits. However, 16-bit and 32-bit character sets are not uncommon – nor are integers too large to fit in 32 bits. For example, many computers now have disks that hold more that *32G* bytes, so 48-bit or 64-bit integers can be useful for representing disk addresses.

To maximize portability, it is wise to be explicit about what implementation-defined features we rely on and to isolate the more subtle examples in clearly marked sections of a program. A typical example of this practice is to present all dependencies on hardware sizes in the form of constants and type definitions in some header file. To support such techniques, the standard library provides **numeric_limits** (§22.2).

Undefined behavior is nastier. A construct is deemed *undefined* by the standard if no reasonable behavior is required by an implementation. Typically, some obvious implementation technique will cause a program using an undefined feature to behave very badly. For example:

```
const int size = 4*1024;
char page[size];

void f()
{
    page[size+size] = 7; // undefined
}
```

Plausible outcomes of this code fragment include overwriting unrelated data and triggering a hardware error/exception. An implementation is not required to choose among plausible outcomes. Where powerful optimizers are used, the actual effects of undefined behavior can become quite unpredictable. If a set of plausible and easily implementable alternatives exist, a feature is deemed implementation-defined rather than undefined.

It is worth spending considerable time and effort to ensure that a program does not use something deemed undefined by the standard. In many cases, tools exist to help do this.

C.3 Character Sets

The examples in this book are written using the U.S. variant of the international 7-bit character set ISO 646-1983 called ASCII (ANSI3.4-1968). This can cause three problems for people who use C++ in an environment with a different character set:

[1] ASCII contains punctuation characters and operator symbols – such as `]`, `{`, and `!` – that are not available in some character sets.

[2] We need a notation for characters that do not have a convenient character representation (e.g., newline and ''the character with value 17'').

[3] ASCII doesn't contain characters, such as – ζ , æ, and Π – that are used for writing languages other than English.

C.3.1 Restricted Character Sets

The ASCII special characters `[`, `]`, `{`, `}`, `|`, and `\` occupy character set positions designated as alphabetic by ISO. In most European national ISO-646 character sets, these positions are occupied by letters not found in the English alphabet. For example, the Danish national character set uses them for the vowels *Æ*, *æ*, *Ø*, *ø*, *Å*, and *å*. No significant amount of text can be written in Danish without them.

A set of trigraphs is provided to allow national characters to be expressed in a portable way using a truly standard minimal character set. This can be useful for interchange of programs, but it doesn't make it easier for people to read programs. Naturally, the long-term solution to this problem is for C++ programmers to get equipment that supports both their native language and C++ well. Unfortunately, this appears to be infeasible for some, and the introduction of new equipment can be a frustratingly slow process. To help programmers stuck with incomplete character sets, C++ provides alternatives:

Keywords		Digraphs		Trigraphs	
and	&&	<%	{	??=	#
and_eq	&=	%>	}	??(	[
bitand	&	<:	[	??<	{
bitor	|	:>	]	??/	\
compl	~	%:	#	??)	]
not	!	%:%:	##	??>	}
or	||			??'	^
or_eq	|=			??!	|
xor	^			??-	~
xor_eq	^=			???	?
not_eq	!=				

Programs using the keywords and digraphs are far more readable than the equivalent programs written using trigraphs. However, if characters such as `{` are not available, trigraphs are necessary for putting ''missing'' characters into strings and character constants. For example, ´`{`´ becomes ´`??<`´.

Some people prefer the keywords such as ***and*** to their traditional operator notation.

C.3.2 Escape Characters

A few characters have standard names that use the backslash \ as an escape character:

Name	ASCII Name	C++ Name
newline	NL (LF)	`\n`
horizontal tab	HT	`\t`
vertical tab	VT	`\v`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
alert	BEL	`\a`
backslash	\	`\\`
question mark	?	`\?`
single quote	'	`\'`
double quote	"	`\"`
octal number	*ooo*	`\`*ooo*
hex number	*hhh*	`\x`*hhh* ...

Despite their appearance, these are single characters.

It is possible to represent a character as a one-, two-, or three-digit octal number (\ followed by octal digits) or as a hexadecimal number (*x* followed by hexadecimal digits). There is no limit to the number of hexadecimal digits in the sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. For example:

Octal	Hexadecimal	Decimal	ASCII
`'\6'`	`'\x6'`	6	ACK
`'\60'`	`'\x30'`	48	`'0'`
`'\137'`	`'\x05f'`	95	`'_'`

This makes it possible to represent every character in the machine's character set and, in particular, to embed such characters in character strings (see §5.2.2). Using any numeric notation for characters makes a program nonportable across machines with different character sets.

It is possible to enclose more than one character in a character literal, for example ´***ab***´. Such uses are archaic, implementation-dependent, and best avoided.

When embedding a numeric constant in a string using the octal notation, it is wise always to use three digits for the number. The notation is hard enough to read without having to worry about whether or not the character after a constant is a digit. For hexadecimal constants, use two digits. Consider these examples:

```
char v1[] = "a\xah\129";      // 6 chars: 'a' '\xa' 'h' '\12' '9' '\0'
char v2[] = "a\xah\127";      // 5 chars: 'a' '\xa' 'h' '\127' '\0'
char v3[] = "a\xad\127";      // 4 chars: 'a' '\xad' '\127' '\0'
char v4[] = "a\xad\0127";     // 5 chars: 'a' '\xad' '\012' '7' '\0'
```

C.3.3 Large Character Sets

A C++ program may be written and presented to the user in character sets that are much richer than the 127 character ASCII set. Where an implementation supports larger character sets, identifiers, comments, character constants, and strings may contain characters such as å, β, and Γ. However, to be portable the implementation must map these characters into an encoding using only characters available to every C++ user. In principle, this translation into the C++ basic source character set (the set used in this book) occurs before the compiler does any other processing. Therefore, it does not affect the semantics of the program.

The standard encoding of characters from large character sets into the smaller set supported directly by C++ is presented as sequences of four or eight hexadecimal digits:

universal-character-name:
    \U *X X X X X X X X*
    \u *X X X X*

Here, ***X*** represents a hexadecimal digit. For example, ***\u1e2b***. The shorter notation ***\uXXXX*** is equivalent to ***\U0000XXXX***. A number of hexadecimal digits different from four or eight is a lexical error.

A programmer can use these character encodings directly. However, they are primarily meant as a way for an implementation that internally uses a small character set to handle characters from a large character set seen by the programmer.

If you rely on special environments to provide an extended character set for use in identifiers, the program becomes less portable. A program is hard to read unless you understand the natural language used for identifiers and comments. Consequently, for programs used internationally it is usually best to stick to English and ASCII.

C.3.4 Signed and Unsigned Characters

It is implementation-defined whether a plain ***char*** is considered signed or unsigned. This opens the possibility for some nasty surprises and implementation dependencies. For example:

```
char c = 255;    // 255 is ''all ones,'' hexadecimal 0xFF
int i = c;
```

What will be the value of *i*? Unfortunately, the answer is undefined. On all implementations I know of, the answer depends on the meaning of the ''all ones'' ***char*** bit pattern when extended into an ***int***. On a SGI Challenge machine, a ***char*** is unsigned, so the answer is ***255***. On a Sun SPARC or an IBM PC, where a ***char*** is signed, the answer is ***-1***. In this case, the compiler might warn about the conversion of the literal ***255*** to the ***char*** value ***-1***. However, C++ does not offer a general mechanism for detecting this kind of problem. One solution is to avoid plain ***char*** and use the specific ***char*** types only. Unfortunately, some standard library functions, such as ***strcmp()***, take plain ***char***s only (§20.4.1).

A ***char*** must behave identically to either a ***signed char*** or an ***unsigned char***. However, the three ***char*** types are distinct, so you can't mix pointers to different ***char*** types. For example:

```
void f(char c, signed char sc, unsigned char uc)
{
    char* pc = &uc;             // error: no pointer conversion
    signed char* psc = pc;      // error: no pointer conversion
    unsigned char* puc = pc;    // error: no pointer conversion
    psc = puc;                  // error: no pointer conversion
}
```

Variables of the three ***char*** types can be freely assigned to each other. However, assigning a too-large value to a signed ***char*** (§C.6.2.1) is still undefined. For example:

```
void f(char c, signed char sc, unsigned char uc)
{
    c = 255;    // undefined if plain chars are signed and have 8 bits

    c = sc;     // ok
    c = uc;     // undefined if plain chars are signed and if uc's value is too large
    sc = uc;    // undefined if uc's value is too large
    uc = sc;    // ok: conversion to unsigned
    sc = c;     // undefined if plain chars are unsigned and if c's value is too large
    uc = c;     // ok: conversion to unsigned
}
```

None of these potential problems occurs if you use plain ***char*** throughout.

C.4 Types of Integer Literals

In general, the type of an integer literal depends on its form, value, and suffix:

- If it is decimal and has no suffix, it has the first of these types in which its value can be represented: ***int***, ***long int***, ***unsigned long int***.
- If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: ***int***, ***unsigned int***, ***long int***, ***unsigned long int***.
- If it is suffixed by ***u*** or ***U***, its type is the first of these types in which its value can be represented: ***unsigned int***, ***unsigned long int***.
- If it is suffixed by ***l*** or ***L***, its type is the first of these types in which its value can be represented: ***long int***, ***unsigned long int***.
- If it is suffixed by ***ul***, ***lu***, ***uL***, ***Lu***, ***Ul***, ***lU***, ***UL***, or ***LU***, its type is ***unsigned long int***.

For example, ***100000*** is of type ***int*** on a machine with 32-bit ***int***s but of type ***long int*** on a machine with 16-bit ***int***s and 32-bit ***long***s. Similarly, ***0XA000*** is of type ***int*** on a machine with 32-bit ***int***s but of type ***unsigned int*** on a machine with 16-bit ***int***s. These implementation dependencies can be avoided by using suffixes: ***100000L*** is of type ***long int*** on all machines and ***0XA000U*** is of type ***unsigned int*** on all machines.

C.5 Constant Expressions

In places such as array bounds (§5.2), case labels (§6.3.2), and initializers for enumerators (§4.8), C++ requires a *constant expression*. A constant expression evaluates to an integral or enumeration constant. Such an expression is composed of literals (§4.3.1, §4.4.1, §4.5.1), enumerators (§4.8), and ***const***s initialized by constant expressions. In a template, an integer template parameter can also be used (§C.13.3). Floating literals (§4.5.1) can be used only if explicitly converted to an integral type. Functions, class objects, pointers, and references can be used as operands to the ***sizeof*** operator (§6.2) only.

Intuitively, constant expressions are simple expressions that can be evaluated by the compiler before the program is linked (§9.1) and starts to run.

C.6 Implicit Type Conversion

Integral and floating-point types (§4.1.1) can be mixed freely in assignments and expressions. Wherever possible, values are converted so as not to lose information. Unfortunately, value-destroying conversions are also performed implicitly. This section provides a description of conversion rules, conversion problems, and their resolution.

C.6.1 Promotions

The implicit conversions that preserve values are commonly referred to as *promotions*. Before an arithmetic operation is performed, *integral promotion* is used to create ***int***s out of shorter integer types. Note that these promotions will *not* promote to ***long*** (unless the operand is a ***wchar_t*** or an enumeration that is already larger than an ***int***). This reflects the original purpose of these promotions in C: to bring operands to the ''natural'' size for arithmetic operations.

The integral promotions are:

- A ***char***, ***signed char***, ***unsigned char***, ***short int***, or ***unsigned short int*** is converted to an ***int*** if ***int*** can represent all the values of the source type; otherwise, it is converted to an ***unsigned int***.
- A ***wchar_t*** (§4.3) or an enumeration type (§4.8) is converted to the first of the following types that can represent all the values of its underlying type: ***int***, ***unsigned int***, ***long***, or ***unsigned long***.
- A bit-field (§C.8.1) is converted to an ***int*** if ***int*** can represent all the values of the bit-field; otherwise, it is converted to ***unsigned int*** if ***unsigned int*** can represent all the values of the bit-field. Otherwise, no integral promotion applies to it.
- A ***bool*** is converted to an ***int***; ***false*** becomes ***0*** and ***true*** becomes ***1***.

Promotions are used as part of the usual arithmetic conversions (§C.6.3).

C.6.2 Conversions

The fundamental types can be converted into each other in a bewildering number of ways. In my opinion, too many conversions are allowed. For example:

```
void f(double d)
{
    char c = d;     // beware: double-precision floating-point to char conversion
}
```

When writing code, you should always aim to avoid undefined behavior and conversions that quietly throw away information. A compiler can warn about many questionable conversions. Fortunately, many compilers actually do.

C.6.2.1 Integral Conversions

An integer can be converted to another integer type. An enumeration value can be converted to an integer type.

If the destination type is ***unsigned***, the resulting value is simply as many bits from the source as will fit in the destination (high-order bits are thrown away if necessary). More precisely, the result is the least unsigned integer congruent to the source integer modulo *2* to the ***n***th, where *n* is the number of bits used to represent the unsigned type. For example:

```
unsigned char uc = 1023; // binary 1111111111: uc becomes binary 11111111; that is, 255
```

If the destination type is ***signed***, the value is unchanged if it can be represented in the destination type; otherwise, the value is implementation-defined:

```
signed char sc = 1023;    // implementation-defined
```

Plausible results are *255* and *-1* (§C.3.4).

A Boolean or enumeration value can be implicitly converted to its integer equivalent (§4.2, §4.8).

C.6.2.2 Floating-Point Conversions

A floating-point value can be converted to another floating-point type. If the source value can be exactly represented in the destination type, the result is the original numeric value. If the source value is between two adjacent destination values, the result is one of those values. Otherwise, the behavior is undefined. For example:

```
float f = FLT_MAX;      // largest float value
double d = f;           // ok: d == f
float f2 = d;           // ok: f2 == f
double d3 = DBL_MAX;    // largest double value
float f3 = d3;          // undefined if FLT_MAX<DBL_MAX
```

C.6.2.3 Pointer and Reference Conversions

Any pointer to an object type can be implicitly converted to a ***void**** (§5.6). A pointer (reference) to a derived class can be implicitly converted to a pointer (reference) to an accessible and unambiguous base (§12.2). Note that a pointer to function or a pointer to member cannot be implicitly converted to a ***void****.

A constant expression (§C.5) that evaluates to *0* can be implicitly converted to any pointer or pointer to member type (§5.1.1). For example:

```
int* p =
            !    !   !     !   !         !
           ! !  !   !     !   !         !
          !  ! !   !     !   !         !
         !    !    ! ! ! ! !   ! ! ! !    ! ! ! !1;
```

A *T** can be implicitly converted to a *const T** (§5.4.1). Similarly, a *T*& can be implicitly converted to a *const T*&.

C.6.2.4 Pointer-to-Member Conversions

Pointers and references to members can be implicitly converted as described in §15.5.1.

C.6.2.5 Boolean Conversions

Pointers, integral, and floating-point values can be implicitly converted to *bool* (§4.2). A nonzero value converts to *true*; a zero value converts to *false*. For example:

```
void f(int* p, int i)
{
    bool is_not_zero = p;       // true if p!=0
    bool b2 = i;                // true if i!=0
}
```

C.6.2.6 Floating-Integral Conversions

When a floating-point value is converted to an integer value, the fractional part is discarded. In other words, conversion from a floating-point type to an integer type truncates. For example, the value of *int(1.6)* is *1*. The behavior is undefined if the truncated value cannot be represented in the destination type. For example:

```
int i = 2.7;          // i becomes 2
char b = 2000.7;      // undefined for 8-bit chars: 2000 cannot be represented as an 8-bit char
```

Conversions from integer to floating types are as mathematically correct as the hardware allows. Loss of precision occurs if an integral value cannot be represented exactly as a value of the floating type. For example,

```
int i = float(1234567890);
```

left *i* with the value *1234567936* on a machine, where both *int*s and *float*s are represented using 32 bits.

Clearly, it is best to avoid potentially value-destroying implicit conversions. In fact, compilers can detect and warn against some obviously dangerous conversions, such as floating to integral and *long int* to *char*. However, general compile-time detection is impractical, so the programmer must be careful. When ‘‘being careful’’ isn’t enough, the programmer can insert explicit checks. For example:

```
class check_failed { };

char checked(int i)
{
        char c = i;                             // warning: not portable (§C.6.2.1)
        if (i != c) throw check_failed();
        return c;
}
void my_code(int i)
{
        char c = checked(i);
        // ...
}
```

To truncate in a way that is guaranteed to be portable requires the use of ***numeric_limits*** (§22.2).

C.6.3 Usual Arithmetic Conversions

These conversions are performed on the operands of a binary operator to bring them to a common type, which is then used as the type of the result:

[1] If either operand is of type ***long double***, the other is converted to ***long double***.
– Otherwise, if either operand is ***double***, the other is converted to ***double***.
– Otherwise, if either operand is ***float***, the other is converted to ***float***.
– Otherwise, integral promotions (§C.6.1) are performed on both operands.
[2] Then, if either operand is ***unsigned long***, the other is converted to ***unsigned long***.
– Otherwise, if one operand is a ***long int*** and the other is an ***unsigned int***, then if a ***long int*** can represent all the values of an ***unsigned int***, the ***unsigned int*** is converted to a ***long int***; otherwise, both operands are converted to ***unsigned long int***.
– Otherwise, if either operand is ***long***, the other is converted to ***long***.
– Otherwise, if either operand is ***unsigned***, the other is converted to ***unsigned***.
– Otherwise, both operands are ***int***.

C.7 Multidimensional Arrays

It is not uncommon to need a vector of vectors, a vector of vector of vectors, etc. The issue is how to represent these multidimensional vectors in C++. Here, I first show how to use the standard library ***vector*** class. Next, I present multidimensional arrays as they appear in C and C++ programs using only built-in facilities.

C.7.1 Vectors

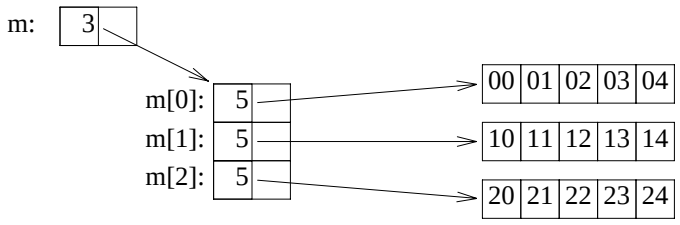
The standard ***vector*** (§16.3) provides a very general solution:

```
vector< vector<int> > m;
```

This creates a vector of vectors of integers that initially contains no elements. We could initialize it to a three-by-five matrix like this:

```
void init_m()
{
    m.resize(3);                        // m now holds 3 empty vectors
    for (int i = 0; i<m.size(); i++) {
        m[i].resize(5);                 // now each of m's vectors holds 5 ints
        for (int j = 0; j<m[i].size(); j++) m[i][j] = 10*i+j;
    }
}
```

or graphically:



Each ***vector*** implementation holds a pointer to its elements plus the number of elements. The elements are typically held in an array. For illustration, I gave each ***int*** an initial value representing its coordinates.

It is not necessary for the ***vector<int>***s in the ***vector< vector<int> >*** to have the same size.

Accessing an element is done by indexing twice. For example, ***m[i][j]*** is the ***j***th element of the ***i***th vector. We can print ***m*** like this:

```
void print_m()
{
    for (int i = 0; i<m.size(); i++) {
        for (int j = 0; j<m[i].size(); j++) cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

which gives:

```
0   1   2   3   4
10  11  12  13  14
20  21  22  23  24
```

C.7.2 Arrays

The built-in arrays are a major source of errors – especially when they are used to build multidimensional arrays. For novices, they are also a major source of confusion. Wherever possible, use ***vector***, ***list***, ***valarray***, ***string***, etc.

Multidimensional arrays are represented as arrays of arrays. A three-by-five array is declared like this:

```
int ma[3][5]; // 3 arrays with 5 ints each
```

For arrays, the dimensions must be given as part of the definition. We can initialize ***ma*** like this:

```
void init_ma()
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) ma[i][j] = 10*i+j;
    }
}
```

or graphically:

ma:	00	01	02	03	04	10	11	12	13	14	20	21	22	23	24

The array ***ma*** is simply 15 ***ints*** that we access as if it were 3 arrays of 5 ***int***s. In particular, there is no single object in memory that is the matrix ***ma*** – only the elements are stored. The dimensions ***3*** and ***5*** exist in the compiler source only. When we write code, it is our job to remember them somehow and supply the dimensions where needed. For example, we might print ***ma*** like this:

```
void print_ma()
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) cout << ma[i][j] << '\t';
        cout << '\n';
    }
}
```

The comma notation used for array bounds in some languages cannot be used in C++ because the comma (,) is a sequencing operator (§6.2.2). Fortunately, most mistakes are caught by the compiler. For example:

```
int bad[3,5];          // error: comma not allowed in constant expression
int good[3][5];        // 3 arrays with 5 ints each
int ouch = good[1,4];  // error: int initialized by int* (good[1,4] means good[4], which is an int*)
int nice = good[1][4];
```

C.7.3 Passing Multidimensional Arrays

Consider defining a function to manipulate a two-dimensional matrix. If the dimensions are known at compile time, there is no problem:

```
void print_m35(int m[3][5])
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

A matrix represented as a multidimensional array is passed as a pointer (rather than copied; §5.3). The first dimension of an array is irrelevant to the problem of finding the location of an element; it simply states how many elements (here `3`) of the appropriate type (here `int[5]`) are present. For example, look at the previous representation of *ma* and note that by our knowing only that the second dimension is `5`, we can locate `ma[i][5]` for any `i`. The first dimension can therefore be passed as an argument:

```
void print_mi5(int m[][5], int dim1)
{
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<5; j++) cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

The difficult case is when both dimensions need to be passed. The ''obvious solution'' simply does not work:

```
void print_mij(int m[][], int dim1, int dim2) // doesn't behave as most people would think
{
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<dim2; j++) cout << m[i][j] << '\t';      // surprise!
        cout << '\n';
    }
}
```

First, the argument declaration `m[][]` is illegal because the second dimension of a multidimensional array must be known in order to find the location of an element. Second, the expression `m[i][j]` is (correctly) interpreted as `*(*(m+i)+j)`, although that is unlikely to be what the programmer intended. A correct solution is:

```
void print_mij(int* m, int dim1, int dim2)
{
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<dim2; j++) cout << m[i*dim2+j] << '\t'; // obscure
        cout << '\n';
    }
}
```

The expression used for accessing the members in `print_mij()` is equivalent to the one the compiler generates when it knows the last dimension.

To call this function, we pass a matrix as an ordinary pointer:

```
int main()
{
    int v[3][5] = { {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24} };

    print_m35(v);
    print_mi5(v,3);
    print_mij(&v[0][0],3,5);
}
```

Note the use of `&v[0][0]` for the last call; `v[0]` would do because it is equivalent, but `v` would be a type error. This kind of subtle and messy code is best hidden. If you must deal directly with multidimensional arrays, consider encapsulating the code relying on it. In that way, you might ease the task of the next programmer to touch the code. Providing a multidimensional array type with a proper subscripting operator saves most users from having to worry about the layout of the data in the array (§22.4.6).

The standard ***vector*** (§16.3) doesn't suffer from these problems.

C.8 Saving Space

When programming nontrivial applications, there often comes a time when you want more memory space than is available or affordable. There are two ways of squeezing more space out of what is available:

[1] Put more than one small object into a byte.

[2] Use the same space to hold different objects at different times.

The former can be achieved by using *fields*, and the latter by using *unions*. These constructs are described in the following sections. Many uses of fields and unions are pure optimizations, and these optimizations are often based on nonportable assumptions about memory layouts. Consequently, the programmer should think twice before using them. Often, a better approach is to change the way data is managed, for example, to rely more on dynamically allocated store (§6.2.6) and less on preallocated (static) storage.

C.8.1 Fields

It seems extravagant to use a whole byte (a ***char*** or a ***bool***) to represent a binary variable – for example, an on/off switch – but a ***char*** is the smallest object that can be independently allocated and addressed in C++ (§5.1). It is possible, however, to bundle several such tiny variables together as *fields* in a ***struct***. A member is defined to be a field by specifying the number of bits it is to occupy. Unnamed fields are allowed. They do not affect the meaning of the named fields, but they can be used to make the layout better in some machine-dependent way:

```
struct PPN {          // R6000 Physical Page Number
    unsigned int PFN : 22;    // Page Frame Number
    int : 3;                  // unused
    unsigned int CCA : 3;     // Cache Coherency Algorithm
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};
```

This example also illustrates the other main use of fields: to name parts of an externally imposed layout. A field must be of an integral or enumeration type (§4.1.1). It is not possible to take the address of a field. Apart from that, however, it can be used exactly like other variables. Note that a ***bool*** field really can be represented by a single bit. In an operating system kernel or in a debugger, the type ***PPN*** might be used like this:

```
void part_of_VM_system(PPN* p)
{
    // ...
    if (p->dirty) { // contents changed
        // copy to disc
        p->dirty = 0;
    }
    // ...
}
```

Surprisingly, using fields to pack several variables into a single byte does not necessarily save space. It saves data space, but the size of the code needed to manipulate these variables increases on most machines. Programs have been known to shrink significantly when binary variables were converted from bit fields to characters! Furthermore, it is typically much faster to access a ***char*** or an ***int*** than to access a field. Fields are simply a convenient shorthand for using bitwise logical operators (§6.2.4) to extract information from and insert information into part of a word.

C.8.2 Unions

A ***union*** is a ***struct*** in which all members are allocated at the same address so that the ***union*** occupies only as much space as its largest member. Naturally, a ***union*** can hold a value for only one member at a time. For example, consider a symbol table entry that holds a name and a value:

```
enum Type { S, I };

struct Entry {
    char* name;
    Type t;
    char* s;    // use s if t==S
    int i;      // use i if t==I
};

void f(Entry* p)
{
    if (p->t == S) cout << p->s;
    // ...
}
```

The members ***s*** and ***i*** can never be used at the same time, so space is wasted. It can be easily recovered by specifying that both should be members of a ***union***, like this:

```
union Value {
    char* s;
    int i;
};
```

The language doesn't keep track of which kind of value is held by a ***union***, so the programmer must still do that:

```
struct Entry {
        char* name;
        Type t;
        Value v;   // use v.s if t==S; use v.i if t==I
};

void f(Entry* p)
{
        if (p->t == S) cout << p->v.s;
        // ...
}
```

Unfortunately, the introduction of the ***union*** forced us to rewrite code to say ***v.s*** instead of plain ***s***. This can be avoided by using an *anonymous union*, which is a union that doesn't have a name and consequently doesn't define a type. Instead, it simply ensures that its members are allocated at the same address:

```
struct Entry {
        char* name;
        Type t;
        union {
                char* s;    // use s if t==S
                int i;      // use i if t==I
        };
};

void f(Entry* p)
{
        if (p->t == S) cout << p->s;
        // ...
}
```

This leaves all code using an ***Entry*** unchanged.

Using a ***union*** so that its value is always read using the member through which it was written is a pure optimization. However, it is not always easy to ensure that a ***union*** is used in this way only, and subtle errors can be introduced through misuse. To avoid errors, one can encapsulate a ***union*** so that the correspondence between a type field and access to the ***union*** members can be guaranteed (§10.6[20]).

Unions are sometimes misused for ''type conversion.'' This misuse is practiced mainly by programmers trained in languages that do not have explicit type conversion facilities, where cheating is necessary. For example, the following ''converts'' an ***int*** to an ***int**** simply by assuming bitwise equivalence:

```
union Fudge {
        int i;
        int* p;
};
```

```
int* cheat(int i)
{
    Fudge a;
    a.i = i;
    return a.p;     // bad use
}
```

This is not really a conversion at all. On some machines, an ***int*** and an ***int**** do not occupy the same amount of space, while on others, no integer can have an odd address. Such use of a ***union*** is dangerous and nonportable, and there is an explicit and portable way of specifying type conversion (§6.2.7).

Unions are occasionally used deliberately to avoid type conversion. One might, for example, use a ***Fudge*** to find the representation of the pointer ***0***:

```
int main()
{
    Fudge foo;
    foo.p = 0;
    cout << "the integer value of the pointer 0 is " << foo.i << '\n';
}
```

C.8.3 Unions and Classes

Many nontrivial ***union***s have some members that are much larger than the most frequently-used members. Because the size of a ***union*** is at least as large as its largest member, space is wasted. This waste can often be eliminated by using a set of derived classes instead of a ***union***.

A class with a constructor, destructor, or copy operation cannot be the type of a ***union*** member (§10.4.12) because the compiler would not know which member to destroy.

C.9 Memory Management

There are three fundamental ways of using memory in C++:

Static memory, in which an object is allocated by the linker for the duration of the program. Global and namespace variables, ***static*** class members (§10.2.4), and ***static*** variables in functions (§7.1.2) are allocated in static memory. An object allocated in static memory is constructed once and persists to the end of the program. It always has the same address. Static objects can be a problem in programs using threads (shared-address space concurrency) because they are shared and require locking for proper access.

Automatic memory, in which function arguments and local variables are allocated. Each entry into a function or a block gets its own copy. This kind of memory is automatically created and destroyed; hence the name automatic memory. Automatic memory is also said ''to be on the stack.'' If you absolutely must be explicit about this, C++ provides the redundant keyword ***auto***.

Free store, from which memory for objects is explicitly requested by the program and where a program can free memory again once it is done with it (using ***new*** and ***delete***). When a program needs more free store, ***new*** requests it from the operating system. Typically, the free

store (also called *dynamic memory* or *the heap*) grows throughout the lifetime of a program because no memory is ever returned to the operating system for use by other programs.

As far as the programmer is concerned, automatic and static storage are used in simple, obvious, and implicit ways. The interesting question is how to manage the free store. Allocation (using ***new***) is simple, but unless we have a consistent policy for giving memory back to the free store manager, memory will fill up – especially for long-running programs.

The simplest strategy is to use automatic objects to manage corresponding objects in free store. Consequently, many containers are implemented as handles to elements stored in the free store (§25.7). For example, an automatic ***String*** (§11.12) manages a sequence of characters on the free store and automatically frees that memory when it itself goes out of scope. All of the standard containers (§16.3, Chapter 17, Chapter 20, §22.4) can be conveniently implemented in this way.

C.9.1 Automatic Garbage Collection

When this regular approach isn't sufficient, the programmer might use a memory manager that finds unreferenced objects and reclaims their memory in which to store new objects. This is usually called *automatic garbage collection*, or simply *garbage collection*. Naturally, such a memory manager is called a *garbage collector*.

The fundamental idea of garbage collection is that an object that is no longer referred to in a program will not be accessed again, so its memory can be safely reused for some new object. For example:

```
void f()
{
    int* p = new int;
    p = 0;
    char* q = new char;
}
```

Here, the assignment ***p=0*** makes the ***int*** unreferenced so that its memory can be used for some other new object. Thus, the ***char*** might be allocated in the same memory as the ***int*** so that ***q*** holds the value that ***p*** originally had.

The standard does not require that an implementation supply a garbage collector, but garbage collectors are increasingly used for C++ in areas where their costs compare favorably to those of manual management of free store. When comparing costs, consider the run time, memory usage, reliability, portability, monetary cost of programming, monetary cost of a garbage collector, and predictability of performance.

C.9.1.1 Disguised Pointers

What should it mean for an object to be unreferenced? Consider:

```
void f()
{
    int* p = new int;
    long i1 = reinterpret_cast<long>(p)&0xFFFF0000;
    long i2 = reinterpret_cast<long>(p)&0x0000FFFF;
    p = 0;
```

```
        // point #1: no pointer to the int exists here

        p = reinterpret_cast<int*>(i1|i2);
        // now the int is referenced again
}
```

Often, pointers stored as non-pointers in a program are called ''disguised pointers.'' In particular, the pointer originally held in ***p*** is disguised in the integers ***i1*** and ***i2***. However, a garbage collector need not be concerned about disguised pointers. If the garbage collector runs at point ***#1***, the memory holding the ***int*** can be reclaimed. In fact, such programs are not guaranteed to work even if a garbage collector is not used because the use of ***reinterpret_cast*** to convert between integers and pointers is at best implementation-defined.

A ***union*** that can hold both pointers and non-pointers presents a garbage collector with a special problem. In general, it is not possible to know whether such a ***union*** contains a pointer. Consider:

```
union U {       // union with both pointer and non-pointer members
        int* p;
        int i;
};

void f(U u, U u2, U u3)
{
        u.p = new int;
        u2.i = 999999;
        u.i = 8;
        // ...
}
```

The safe assumption is that any value that appears in such a ***union*** is a pointer value. A clever garbage collector can do somewhat better. For example, it may notice that (for a given implementation) ***int***s are not allocated with odd addresses and that no objects are allocated with an address as low as ***8***. Noticing this will save the garbage collector from having to assume that objects containing locations ***999999*** and ***8*** are used by ***f***().

C.9.1.2 Delete

If an implementation automatically collects garbage, the ***delete*** and ***delete***[] operators are no longer needed to free memory for potential reuse. Thus, a user relying on a garbage collector could simply refrain from using these operators. However, in addition to freeing memory, ***delete*** and ***delete***[] invoke destructors.

In the presence of a garbage collector,

```
delete p;
```

invokes the destructor for the object pointed to by ***p*** (if any). However, reuse of the memory can be postponed until it is collected. Recycling lots of objects at once can help limit fragmentation (§C.9.1.4). It also renders harmless the otherwise serious mistake of deleting an object twice in the important case where the destructor simply deletes memory.

As always, access to an object after it has been deleted is undefined.

C.9.1.3 Destructors

When an object is about to be recycled by a garbage collector, two alternatives exist:

[1] Call the destructor (if any) for the object.

[2] Treat the object as raw memory (don't call its destructor).

By default, a garbage collector should choose option (2) because objects created using ***new*** and never ***delete***d are never destroyed. Thus, one can see a garbage collector as a mechanism for simulating an infinite memory.

It is possible to design a garbage collector to invoke the destructors for objects that have been specifically ''registered'' with the collector. However, there is no standard way of ''registering'' objects. Note that it is always important to destroy objects in an order that ensures that the destructor for one object doesn't refer to an object that has been previously destroyed. Such ordering isn't easily achieved by a garbage collector without help from the programmer.

C.9.1.4 Memory Fragmentation

When a lot of objects of varying sizes are allocated and freed, the memory *fragments*. That is, much of memory is consumed by pieces of memory that are too small to use effectively. The reason is that a general allocator cannot always find a piece of memory of the exact right size for an object. Using a slightly larger piece means that a smaller fragment of memory remains. After running a program for a while with a naive allocator, it is not uncommon to find half the available memory taken up with fragments too small ever to get reused.

Several techniques exist for coping with fragmentation. The simplest is to request only larger chunks of memory from the allocator and use each such chunk for objects of the same size (§15.3, §19.4.2). Because most allocations and deallocations are of small objects of types such as tree nodes, links, etc., this technique can be very effective. An allocator can sometimes apply similar techniques automatically. In either case, fragmentation is further reduced if all of the larger ''chunks'' are of the same size (say, the size of a page) so that they themselves can be allocated and reallocated without fragmentation.

There are two main styles of garbage collectors:

[1] A *copying collector* moves objects in memory to compact fragmented space.

[2] A *conservative collector* allocates objects to minimize fragmentation.

From a C++ point of view, conservative collectors are preferable because it is very hard (probably impossible in real programs) to move an object and modify all pointers to it correctly. A conservative collector also allows C++ code fragments to coexist with code written in languages such as C. Traditionally, copying collectors have been favored by people using languages (such as Lisp and Smalltalk) that deal with objects only indirectly through unique pointers or references. However, modern conservative collectors seem to be at least as efficient as copying collectors for larger programs, in which the amount of copying and the interaction between the allocator and a paging system become important. For smaller programs, the ideal of simply never invoking the collector is often achievable – especially in C++, where many objects are naturally automatic.

C.10 Namespaces

This section presents minor points about namespaces that look like technicalities, yet frequently surface in discussions and in real code.

C.10.1 Convenience vs. Safety

A *using-declaration* adds a name to a local scope. A *using-directive* does not; it simply renders names accessible in the scope in which they were declared. For example:

```
namespace X {
        int i, j, k;
}

int k;

void f1()
{
        int i = 0;
        using namespace X; // make names from X accessible
        i++;               // local i
        j++;               // X::j
        k++;               // error: X::k or global k ?
        ::k++;             // the global k
        X::k++;            // X's k
}

void f2()
{
        int i = 0;
        using X::i;     // error: i declared twice in f2()
        using X::j;
        using X::k;     // hides global k

        i++;
        j++;            // X::j
        k++;            // X::k
}
```

A locally declared name (declared either by an ordinary declaration or by a *using-declaration*) hides nonlocal declarations of the same name, and any illegal overloadings of the name are detected at the point of declaration.

Note the ambiguity error for `k++` in `f1()`. Global names are not given preference over names from namespaces made accessible in the global scope. This provides significant protection against accidental name clashes, and – importantly – ensures that there are no advantages to be gained from polluting the global namespace.

When libraries declaring *many* names are made accessible through *using-directive*s, it is a significant advantage that clashes of unused names are not considered errors.

The global scope is just another namespace. The global namespace is odd only in that you don't have to mention its name in an explicit qualification. That is, `::k` means ''look for `k` in the global namespace and in namespaces mentioned in *using-directive*s in the global namespace,''

whereas *X*::*k* means ''the *k* declared in namespace *X* and namespaces mentioned in *using-directive*s in *X*'' (§8.2.8).

I hope to see a radical decrease in the use of global names in new programs using namespaces compared to traditional C and C++ programs. The rules for namespaces were specifically crafted to give no advantages to a ''lazy'' user of global names over someone who takes care not to pollute the global scope.

C.10.2 Nesting of Namespaces

One obvious use of namespaces is to wrap a complete set of declarations and definitions in a separate namespace:

```
namespace X {
    // all my declarations
}
```

The list of declarations will, in general, contain namespaces. Thus, nested namespaces are allowed. This is allowed for practical reasons, as well as for the simple reason that constructs ought to nest unless there is a strong reason for them not to. For example:

```
void h();

namespace X {
    void g();
    // ...
    namespace Y {
        void f();
        void ff();
        // ...
    }
}
```

The usual scope and qualification rules apply:

```
void X::Y::ff()
{
    f(); g(); h();
}

void X::g()
{
    f();        // error: no f() in X
    Y::f();     // ok
}

void h()
{
    f();            // error: no global f()
    Y::f();         // error: no global Y
    X::f();         // error: no f() in X
    X::Y::f();      // ok
}
```

C.10.3 Namespaces and Classes

A namespace is a named scope. A class is a type defined by a named scope that describes how objects of that type can be created and used. Thus, a namespace is a simpler concept than a class and ideally a class would be defined as a namespace with a few extra facilities included. This is almost the case. A namespace is open (§8.2.9.3), but a class is closed. This difference stems from the observation that a class needs to define the layout of an object and that is best done in one place. Furthermore, *using-declaration*s and *using-directive*s can be applied to classes only in a very restricted way (§15.2.2).

Namespaces are preferred over classes when all that is needed is encapsulation of names. In this case, the class apparatus for type checking and for creating objects is not needed; the simpler namespace concept suffices.

C.11 Access Control

This section presents a few technical examples illustrating access control to supplement those presented in §15.3.

C.11.1 Access to Members

Consider:

```
class X {
// private by default:
      int priv;
protected:
      int prot;
public:
      int publ;
      void m();
};
```

The member `X::m()` has unrestricted access:

```
void X::m()
{
      priv = 1;   // ok
      prot = 2;   // ok
      publ = 3;   // ok
}
```

A member of a derived class has access to public and protected members (§15.3):

```
class Y : public X {
      void mderived();
};
```

```
void Y::mderived()
{
    priv = 1;   // error: priv is private
    prot = 2;   // ok: prot is protected and mderived() is a member of the derived class Y
    publ = 3;   // ok: publ is public
}
```

A global function can access only the public members:

```
void f(Y* p)
{
    p->priv = 1;    // error: priv is private
    p->prot = 2;    // error: prot is protected and f() is not a friend or a member of X or Y
    p->publ = 3;    // ok: publ is public
}
```

C.11.2 Access to Base Classes

Like a member, a base class can be declared ***private***, ***protected***, or ***public***. Consider:

```
class X {
public:
    int a;
    // ...
};

class Y1 : public X { };
class Y2 : protected X { };
class Y3 : private X { };
```

Because ***X*** is a public base of ***Y1***, any function can (implicitly) convert a ***Y1**** to an ***X**** where needed just as it can access the public members of class ***X***. For example:

```
void f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;    // ok: X is a public base class of Y1
    py1->a = 7;     // ok

    px = py2;       // error: X is a protected base of Y2
    py2->a = 7;     // error

    px = py3;       // error: X is a private base of Y3
    py3->a = 7;     // error
}
```

Consider:

```
class Y2 : protected X { };
class Z2 : public Y2 { void f(Y1*, Y2*, Y3*); };
```

Because ***X*** is a protected base of ***Y2***, only members and friends of ***Y2*** and members and friends of ***Y2***'s derived classes (e.g., ***Z2***) can (implicitly) convert a ***Y2**** to an ***X**** where needed, just as they can access the public and protected members of class ***X***. For example:

```
void Z2::f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;    // ok: X is a public base class of Y1
    py1->a = 7;     // ok

    px = py2;       // ok: X is a protected base of Y2, and Z2 is derived from Y2
    py2->a = 7;     // ok

    px = py3;       // error: X is a private base of Y3
    py3->a = 7;     // error
}
```

Consider finally:

```
class Y3 : private X { void f(Y1*, Y2*, Y3*); };
```

Because *X* is a private base of *Y3*, only members and friends of *Y3* can (implicitly) convert a *Y3** to an *X** where needed, just as they can access the public and protected members of class *X*. For example:

```
void Y3::f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;    // ok: X is a public base class of Y1
    py1->a = 7;     // ok

    px = py2;       // error: X is a protected base of Y2
    py2->a = 7;     // error

    px = py3;       // ok: X is a private base of Y3, and Y3::f() is a member of Y3
    py3->a = 7;     // ok
}
```

C.11.3 Access to Member Class

The members of a member class have no special access to members of an enclosing class. Similarly members of an enclosing class have no special access to members of a nested class; the usual access rules (§10.2.2) shall be obeyed. For example:

```
class Outer {
    typedef int T;
    int i;
public:
    int i2;
    static int s;

    class Inner {
        int x;
        T y; // error: Outer::T is private
    public:
        void f(Outer* p, int v);
    };
```

```
    int g(Inner* p);
};

void Outer::Inner::f(Outer* p, int v)
{
    p->i = v;       // error: Outer::i is private
    p->i2 = v;      // ok: Outer::i2 is public
}

int Outer::g(Inner* p)
{
    p->f(this,2);   // ok: Inner::f() is public
    return p->x;    // error: Inner::x is private
}
```

However, it is often useful to grant a member class access to its enclosing class. This can be done by making the member a ***friend***. For example:

```
class Outer {
    typedef int T;
    int i;
public:
    class Inner;            // forward declaration of member class
    friend class Inner;     // grant access to Outer::Inner

    class Inner {
        int x;
        T y;        // ok: Inner is a friend
    public:
        void f(Outer* p, int v);
    };
};

void Outer::Inner::f(Outer* p, int v)
{
    p->i = v;  // ok: Inner is a friend
}
```

C.11.4 Friendship

Friendship is neither inherited nor transitive. For example:

```
class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};
```

```
class C {
    void f(A* p)
    {
        p->a++;   // error: C is not a friend of A, despite being a friend of a friend of A
    }
};

class D : public B {
    void f(A* p)
    {
        p->a++;   // error: D is not a friend of A, despite being derived from a friend of A
    }
};
```

C.12 Pointers to Data Members

Naturally, the notion of pointer to member (§15.5) applies to data members and to member functions with arguments and return types. For example:

```
struct C {
    char* val;
    int i;
    void print(int x) { cout << val << x << '\n'; }
    void f1();
    int f2();
    C(char* v) { val = v; }
};

typedef void (C::*PMFI)(int);       // pointer to member function of C taking an int
typedef char* C::*PM;               // pointer to char* data member of C

void f(C& z1, C& z2)
{
    C* p = &z2;
    PMFI pf = &C::print;
    PM pm = &C::val;

    z1.print(1);
    (z1.*pf)(2);
    z1.*pm = "nv1 ";
    p->*pm = "nv2 ";
    z2.print(3);
    (p->*pf)(4);

    pf = &C::f1;    // error: return type mismatch
    pf = &C::f2;    // error: argument type mismatch
    pm = &C::i;     // error: type mismatch
    pm = pf;        // error: type mismatch
}
```

The type of a pointer to function is checked just like any other type.

C.13 Templates

A class template specifies how a class can be generated given a suitable set of template arguments. Similarly, a function template specifies how a function can be generated given a suitable set of template arguments. Thus, a template can be used to generate types and executable code. With this expressive power comes some complexity. Most of this complexity relates to the variety of contexts involved in the definition and use of templates.

C.13.1 Static Members

A class template can have ***static*** members. Each class generated from the template has its own copy of the static members. Static members must be separately defined and can be specialized. For example:

```
template<class T> class X {
    // ...
    static T def_val;
    static T* new_X(T a = def_val);
};

template<class T> T X<T>::def_val(0,0);
template<class T> T* X<T>::new_X(T a) { /* ... */ }

template<> int X<int>::def_val<int> = 0;
template<> int* X<int>::new_X<int>(int i) { /* ... */ }
```

If you want to share an object or function among all members of every class generated from a template, you can place it in a non-templatized base class. For example:

```
struct B {
    static B* nil;    // to be used as common null pointer for every class derived from B
};

template<class T> class X : public B {
    // ...
};

B* B::nil = 0;
```

C.13.2 Friends

Like other classes, a template class can have friends. For example, comparison operators are typically friends, so we can rewrite class ***Basic_ops*** from §13.6 like this:

```
template <class C> class Basic_ops { // basic operators on containers
    friend bool operator==(const C&, const C&); // compare elements
    friend bool operator!=(const C&, const C&);
    // ...
};
```

```
template<class T> class Math_container : public Basic_ops< Math_container<T> > {
    // ...
};
```

Like a member, a friend declared within a template is itself a template and is defined using the template parameters of its class. For example:

```
template <class C> bool operator==(const C& a, const C& b)
{
    if (a.size() != b.size()) return false;
    for (int i = 0; i<a.size(); ++i)
        if (a[i] != b[i]) return false;
    return true;
}
```

Friends do not affect the scope in which the template class is defined, nor do they affect the scope in which the template is used. Instead, friend functions and operators are found using a lookup based on their argument types (§11.2.4, §11.5.1). Like a member function, a friend function is instantiated (§C.13.9.1) only if it is called.

C.13.3 Templates as Template Parameters

Sometimes it is useful to pass templates – rather than classes or objects – as template arguments. For example:

```
template<class T, template<class> class C> class Xrefd {
    C<T> mems;
    C<T*> refs;
    // ...
};

Xrefd<Entry, vector> x1;    // store cross references for Entries in a vector

Xrefd<Record, set> x2;      // store cross references for Records in a set
```

To use a template as a template parameter, you specify its required arguments. The template parameters of the template parameter need to be known in order to use the template parameter. The point of using a template as a template parameter is usually that we want to instantiate it with a variety of argument types (such as ***T*** and ***T**** in the previous example). That is, we want to express the member declarations of a template in terms of another template, but we want that other template to be a parameter so that it can be specified by users.

The common case in which a template needs a container to hold elements of its own argument type is often better handled by passing the container type (§13.6, §17.3.1).

Only class templates can be template arguments.

C.13.4 Deducing Function Template Arguments

A compiler can deduce a type template argument, ***T*** or ***TT***, and a non-type template argument, ***I***, from a template function argument with a type composed of the following constructs:

T	*const T*	*volatile T*
*T**	*T&*	*T*[*constant_expression*]
type[*I*]	*class_template_name<T>*	*class_template_name<I>*
TT<T>	*T<I>*	*T<>*
*T type::**	*T T::**	*type T::**
T (*)(args)	*type (T::*)(args)*	*T (type::*)(args)*
type (type::)(args_TI)*	*T (T::*)(args_TI)*	*type (T::*)(args_TI)*
T (type::)(args_TI)*	*type (*)(args_TI)*	

Here, ***args_TI*** is a parameter list from which a ***T*** or an ***I*** can be determined by recursive application of these rules and ***args*** is a parameter list that does not allow deduction. If not all parameters can be deduced in this way, a call is ambiguous. For example:

```
template<class T, class U> void f(const T*, U(*)(U));

int g(int);

void h(const char* p)
{
    f(p,g);    // T is char, U is int
    f(p,h);    // error: can't deduce U
}
```

Looking at the arguments of the first call of `f()`, we easily deduce the template arguments. Looking at the second call of `f()`, we see that `h()` doesn't match the pattern `U(*)(U)` because `h()`'s argument and return types differ.

If a template parameter can be deduced from more than one function argument, the same type must be the result of each deduction. Otherwise, the call is an error. For example:

```
template<class T> void f(T i, T* p);

void g(int i)
{
    f(i,&i);              // ok
    f(i,"Remember!");     // error, ambiguous: T is int or T is char?
}
```

C.13.5 Typename and Template

To make generic programming easier and more general, the standard library containers provide a set of standard functions and types (§16.3.1). For example:

```
template<class T> class vector {
public:
    typedef T value_type;
    typedef T* iterator;

    iterator begin();
    iterator end();

    // ...
};
```

```
template<class T> class list {
    class link {
        // ...
    };
public:
    typedef T value_type;
    typedef link* iterator;

    iterator begin();
    iterator end();

    // ...
};
```

This allows us to write:

```
void f1(vector<T>& v)
{
    vector<T>::iterator i = v.begin();
    // ...
}

void f2(list<T>& v)
{
    list<T>::iterator i = v.begin();
    // ...
}
```

However, this does not allow us to write:

```
template<class C> void f4(C& v)
{
    C::iterator i = v.begin(); // error
    // ...
}
```

Unfortunately, the compiler isn't required to be psychic, so it doesn't know that *C::iterator* is the name of a type. In the previous example, the compiler could look at the declaration of *vector<>* to determine that the *iterator* in *vector<T>::iterator* was a type. That is not possible when the qualifier is a type parameter. Naturally, a compiler could postpone all checking until instantiation time where all information is available and could then accept such examples. However, that would be a nonstandard language extension.

Consider an example stripped of clues as to its meaning:

```
template<class T> void f5(T& v)
{
    T::x(y); // error?
}
```

Is *T::x* a function called with a nonlocal variable *y* as its argument? Or, are we declaring a variable *y* with the type *T::x* perversely using redundant parentheses? We could imagine a context in which *X::x(y)* was a function call and *Y::x(y)* was a declaration.

The resolution is simple: unless otherwise stated, an identifier is assumed to refer to something that is not a type or a template. If we want to state that something should be treated as a type, we can do so using the ***typename*** keyword:

```
template<class C> void f4(C& v)
{
    typename C::iterator i = v.begin();
    // ...
}
```

The ***typename*** keyword can be placed in front of a qualified name to state that the entity named is a type. In this, it resembles ***struct*** and ***class***.

The ***typename*** keyword can also be used as an alternative to ***class*** in template declarations. For example:

```
template<typename T> void f(T);
```

Being an indifferent typist and always short of screen space, I prefer the shorter:

```
template<class T> void f(T);
```

C.13.6 Template as a Qualifier

The need for the ***typename*** qualifier arises because we can refer both to members that are types and to members that are non-types. We can also have members that are templates. In rare cases, the need to distinguish the name of a template member from other member names can arise. Consider a possible interface to a general memory manager:

```
class Memory { // some Allocator
public:
    template<class T> T* get_new();
    template<class T> void release(T&);
    // ...
};

template<class Allocator> void f(Allocator& m)
{
    int* p1 = m.get_new<int>();                 // syntax error: int after less-than operator
    int* p2 = m.template get_new<int>();        // explicit qualification
    // ...
    m.release(p1); // template argument deduced: no explicit qualification needed
    m.release(p2);
}
```

Explicit qualification of ***get_new()*** is necessary because its template parameter cannot be deduced. In this case, the ***template*** prefix must be used to inform the compiler (and the human reader) that ***get_new*** is a member template so that explicit qualification with the desired type of element is possible. Without the qualification with ***template***, we would get a syntax error because the < would be assumed to be a less-than operator. The need for qualification with ***template*** is rare because most template parameters are deduced.

C.13.7 Instantiation

Given a template definition and a use of that template, it is the implementation's job to generate correct code. From a class template and a set of template arguments, the compiler needs to generate the definition of a class and the definitions of those of its member functions that were used. From a template function, a function needs to be generated. This process is commonly called *template instantiation*.

The generated classes and functions are called *specializations*. When there is a need to distinguish between generated specializations and specializations explicitly written by the programmer (§13.5), these are referred to as *generated specializations* and *explicit specializations*, respectively. An explicit specialization is sometimes referred to as a *user-defined specialization*, or simply a *user specialization.*

To use templates in nontrivial programs, a programmer must understand how names used in a template definition are bound to declarations and how source code can be organized (§13.7).

By default, the compiler generates classes and functions from the templates used in accordance with the name-binding rules (§C.13.8). That is, a programmer need not state explicitly which versions of which templates must be generated. This is important because it is not easy for a programmer to know exactly which versions of a template are needed. Often, templates that the programmer hasn't even heard of are used in the implementation of libraries, and sometimes templates that the programmer does know of are used with unknown template argument types. In general, the set of generated functions needed can be known only by recursive examination of the templates used in application code libraries. Computers are better suited than humans for doing such analysis.

However, it is sometimes important for a programmer to be able to state specifically where code should be generated from a template (§C.13.10). By doing so, the programmer gains detailed control over the context of the instantiation. In most compilation environments, this also implies control over exactly when that instantiation is done. In particular, explicit instantiation can be used to force compilation errors to occur at predictable times rather than occurring whenever an implementation determines the need to generate a specialization. A perfectly predictable build process is essential to some users.

C.13.8 Name Binding

It is important to define template functions so that they have as few dependencies as possible on nonlocal information. The reason is that a template will be used to generate functions and classes based on unknown types and in unknown contexts. Every subtle context dependency is likely to surface as a debugging problem for some programmer – and that programmer is unlikely to want to know the implementation details of the template. The general rule of avoiding global names as far as possible should be taken especially seriously in template code. Thus, we try to make template definitions as self-contained as possible and to supply much of what would otherwise have been global context in the form of template parameters (e.g., traits; §13.4, §20.2.1).

However, some nonlocal names must be used. In particular, it is more common to write a set of cooperating template functions than to write just one self-contained function. Sometimes, such functions can be class members, but not always. Sometimes, nonlocal functions are the best choice. Typical examples of that are ***sort***`()`'s calls to ***swap***`()` and ***less***`()` (§13.5.2). The standard library algorithms provide a large-scale example (Chapter 18).

Operations with conventional names and semantics, such as +, *, [], and *sort*(), are another source of nonlocal name use in a template definition. Consider:

```
#include<vector>

bool tracing;

// ...

template<class T> T sum(std::vector<T>& v)
{
        T t = 0;
        if (tracing) cerr << "sum(" << &v << ")\n";
        for (int i = 0; i<v.size(); i++) t = t + v[i];
        return t;
}

// ...

#include<quad.h>

void f(std::vector<Quad>& v)
{
        Quad c = sum(v);
}
```

The innocent-looking template function *sum*() depends on the + operator. In this example, + is defined in <*quad.h*>:

```
Quad operator+(Quad,Quad);
```

Importantly, nothing related to complex numbers is in scope when *sum*() is defined and the writer of *sum*() cannot be assumed to know about class *Quad*. In particular, the + may be defined later than *sum*() in the program text, and even later in time.

The process of finding the declaration for each name explicitly or implicitly used in a template is called *name binding*. The general problem with template name binding is that three contexts are involved in a template instantiation and they cannot be cleanly separated:

[1] The context of the template definition
[2] The context of the argument type declaration
[3] The context of the use of the template

C.13.8.1 Dependent Names

When defining a function template, we want to assure that enough context is available for the template definition to make sense in terms of its actual arguments without picking up ''accidental'' stuff from the environment of a point of use. To help with this, the language separates names used in a template definition into two categories:

[1] Names that depend on a template argument. Such names are bound at some point of instantiation (§C.13.8.3). In the *sum*() example, the definition of + can be found in the instantiation context because it takes operands of the template argument type.

[2] Names that don't depend on a template argument. Such names are bound at the point of definition of the template (§C.13.8.2). In the ***sum*()** example, the template ***vector*** is defined in the standard header ***<vector>*** and the Boolean ***tracing*** is in scope when the definition of ***sum*()** is encountered by the compiler.

The simplest definition of ''***N*** depends on a template parameter ***T***'' would be ''***N*** is a member of ***T***.'' Unfortunately, this doesn't quite suffice; addition of ***Quad***s (§C.13.8) is a counter-example. Consequently, a function call is said to *depend on* a template argument if and only if one of these conditions hold:

[1] The type of the actual argument depends on a template parameter ***T*** according to the type deduction rules (§13.3.1). For example, ***f*(*T*(1))**, ***f*(*t*)**, ***f*(*g*(*t*))**, and ***f*(&*t*)**, assuming that ***t*** is a ***T***.

[2] The function called has a formal parameter that depends on ***T*** according to the type deduction rules (§13.3.1). For example, ***f*(*T*)**, ***f*(*list*<*T*>&)**, and ***f*(*const T**)**.

Basically, the name of a function called is dependent if it is obviously dependent by looking at its arguments or at its formal parameters.

A call that by coincidence has an argument that matches an actual template parameter type is not dependent. For example:

```
template<class T> T f(T a)
{
    return g(1);    // error: no g() in scope and g(1) doesn't depend on T
}

void g(int);

int z = f(2);
```

It doesn't matter that for the call ***f*(2)**, ***T*** happens to be ***int*** and ***g*()**'s argument just happens to be an ***int***. Had ***g*(1)** been considered dependent, its meaning would have been most subtle and mysterious to the reader of the template definition. If a programmer wants ***g*(*int*)** to be called, ***g*(*int*)**'s definition should be placed before the definition of ***f*()** so that ***g*(*int*)** is in scope when ***f*()** is analyzed. This is exactly the same rule as for non-template function definitions.

Note that only names of functions used in calls can be dependent names according to this definition. Names of variables, class members, types, etc., in a template definition must be declared (possibly in terms of template parameters) before they are used.

C.13.8.2 Point of Definition Binding

When the compiler sees a template definition, it determines which names are dependent (§C.13.8.1). If a name is dependent, looking for its declaration must be postponed until instantiation time (§C.13.8.3).

Names that do not depend on a template argument must be in scope (§4.9.4) at the point of definition. For example:

```
int x;
```

```
template<class T> T f(T a)
{
    x++;        // ok
    y++;        // error: no y in scope, and y doesn't depend on T
    return a;
}

int y;

int z = f(2);
```

If a declaration is found, that declaration is used even if a ''better'' declaration might be found later. For example:

```
void g(double);

template<class T> class X : public T {
public:
    void f() { g(2); }    // call g(double);
    // ...
};

void g(int);

class Z { };

void h(X<Z> x)
{
    x.f();
}
```

When a definition for `X<Z>::f()` is generated, `g(int)` is not considered because it is declared after `X`. It doesn't matter that `X` is not used until after the declaration of `g(int)`. Also, a call that isn't dependent cannot be hijacked in a base class:

```
class Y { public: void g(int); };

void h(X<Y> x)
{
    x.f();
}
```

Again, `X<Y>::f()` will call `g(double)`. If the programmer had wanted the `g()` from the base class `T` to be called, the definition of `f()` should have said so:

```
template<class T> class XX : public T {
    void f() { T::g(2); }    // calls T::g()
    // ...
};
```

This is, of course, an application of the rule of thumb that a template definition should be as self-contained as possible.

C.13.8.3 Point of Instantiation Binding

Each use of a template for a given set of template arguments defines a point of instantiation. That point is in the nearest global or namespace scope enclosing its use, just before the declaration that contains that use. For example:

```
template<class T> void f(T a) { g(a); }

void g(int);

void h()
{
    extern g(double);
    f(2);
}
```

Here, the point of instantiation for `f<int>()` is just before `h()`, so the `g()` called in `f()` is the global `g(int)` rather than the local `g(double)`. The definition of ''instantiation point'' implies that a template parameter can never be bound to a local name or a class member. For example:

```
void f()
{
    struct X { /* ... */ };      // local structure
    vector<X> v;                 // error: cannot use local structure as template parameter
    // ...
}
```

Nor can an unqualified name used in a template ever be bound to a local name. Finally, even if a template is first used within a class, unqualified names used in the template will not be bound to members of that class. Ignoring local names is essential to prevent a lot of nasty macro-like behavior. For example:

```
template<class T> void sort(vector<T>& v)
{
    sort(v.begin(),v.end());     // use standard library sort()
}

class Container {
    vector<int> v;   // elements
    // ...
public:
    void sort()      // sort elements
    {
        sort(v);     // invokes sort(vector<int>&) rather than Container::sort()
    }
    // ...
};
```

If the point of instantiation for a template defined in a namespace is in another namespace, names from both namespaces are available for name binding. As always, overload resolution is used to choose between names from different namespaces (§8.2.9.2).

Note that a template used several times with the same set of template arguments has several points of instantiation. If the bindings of independent names differ, the program is illegal. However, this is a difficult error for an implementation to detect, especially if the points of instantiation are in different translation units. It is best to avoid subtleties in name binding by minimizing the use of nonlocal names in templates and by using header files to keep use contexts consistent.

C.13.8.4 Templates and Namespaces

When a function is called, its declaration can be found even if it is not in scope, provided it is declared in the same namespace as one of its arguments (§8.2.6). This is very important for functions called in template definitions because it is the mechanism by which dependent functions are found during instantiation.

A template specialization may be generated at any point of instantiation (§C.13.8.3), any point subsequent to that in a translation unit, or in a translation unit specifically created for generating specializations. This reflects three obvious strategies an implementation can use for generating specializations:

[1] Generate a specialization the first time a call is seen.
[2] At the end of a translation unit, generate all specializations needed for that translation unit.
[3] Once every translation unit of a program has been seen, generate all specializations needed for the program.

All three strategies have strengths and weaknesses, and combinations of these strategies are also possible.

In any case, the binding of independent names is done at a point of template definition. The binding of dependent names is done by looking at

[1] the names in scope at the point where the template is defined, plus
[2] the names in the namespace of an argument of a dependent call (global functions are considered in the namespace of built-in types).

For example:

```
namespace N {
    class A { /* ... */ };

    char f(A);
}

char f(int);

template<class T> char g(T t) { return f(t); }

char c = g(N::A());          // causes N::f(N::A) to be called
```

Here, $f(t)$ is clearly dependent, so we can't bind f to $f(N::A)$ or $f(int)$ at the point of definition. To generate a specialization for $g<N::A>(N::A)$, the implementation looks in namespace N for functions called $f()$ and finds $N::f(N::A)$.

A program is illegal, if it is possible to construct two different meanings by choosing different points of instantiation or different contents of namespaces at different possible contexts for generating the specialization. For example:

```
namespace N {
    class A { /* ... */ };

    char f(A,int);
}

template<class T, class T2> char g(T t, T2 t2) { return f(t,t2); }

char c = g(N::A(),'a');      // error (alternative resolutions of f(t) possible)

namespace N {                // add to namespace N (§8.2.9.3)
    void f(A,char);
}
```

We could generate the specialization at the point of instantiation and get `f(N::A,int)` called. Alternatively, we could wait and generate the specialization at the end of the translation unit and get `f(N::A,char)` called. Consequently, the call `g(N::A(),'a')` is an error.

It is sloppy programming to call an overloaded function in between two of its declarations. Looking at a large program, a programmer would have no reason to suspect a problem. In this particular case, a compiler could catch the ambiguity. However, similar problems can occur in separate translation units, and then detection becomes much harder. An implementation is not obliged to catch problems of this kind.

Most problems with alternative resolutions of function calls involve built-in types. Consequently, most remedies rely on more-careful use of arguments of built-in types.

As usual, use of global functions can make matters worse. The global namespace is considered the namespace associated with built-in types, so global functions can be used to resolve dependent calls that take built-in types. For example:

```
int f(int);

template<class T> T g(T t) { return f(t); }

char c = g('a');     // error: alternative resolutions of f(t) are possible

char f(char);
```

We could generate the specialization `g<char>(char)` at the point of instantiation and get `f(int)` called. Alternatively, we could wait and generate the specialization at the end of the translation unit and get `f(char)` called. Consequently, the call `g('a')` is an error.

C.13.9 When Is a Specialization Needed?

It is necessary to generate a specialization of a class template only if the class' definition is needed. In particular, to declare a pointer to some class, the actual definition of a class is not needed. For example:

```
class X;
X* p;       // ok: no definition of X needed
X a;        // error: definition of X needed
```

When defining template classes, this distinction can be crucial. A template class is *not* instantiated unless its definition is actually needed. For example:

```
template<class T> class Link {
    Link* suc; // ok: no definition of Link needed (yet)
    // ...
};

Link<int>* pl;  // no instantiation of Link<int> needed

Link<int> lnk;  // now we need to instantiate Link<int>
```

The point of instantiation is where a definition is first needed.

C.13.9.1 Template Function Instantiation

An implementation instantiates a template function only if that function has been used. In particular, instantiation of a class template does not imply the instantiation of all of its members or even of all of the members defined in the template class declaration. This allows the programmer an important degree of flexibility when defining a template class. Consider:

```
template<class T> class List {
    // ...
    void sort();
};

class Glob { /* no comparison operators */ };

void f(List<Glob>& lb, List<string>& ls)
{
    ls.sort();
    // use operations on lb, but not lb.sort()
}
```

Here, ***List<string>::sort()*** is instantiated, but ***List<Glob>::sort()*** isn't. This both reduces the amount of code generated and saves us from having to redesign the program. Had ***List<Glob>::sort()*** been generated, we would have had to either add the operations needed by ***vector::sort()*** to ***Glob***, redefine ***sort()*** so that it wasn't a member of ***List***, or use some other container for ***Glob***s.

C.13.10 Explicit Instantiation

An explicit instantiation request is a declaration of a specialization prefixed by the keyword ***template*** (not followed by <):

```
template class vector<int>;                  // class
template int& vector<int>::operator[](int);  // member
template int convert<int,double>(double);    // function
```

A template declaration starts with ***template<***, whereas plain ***template*** starts an instantiation request. Note that ***template*** prefixes a complete declaration; just stating a name is not sufficient:

```
template vector<int>::operator[];    // syntax error
template convert<int,double>;        // syntax error
```

As in template function calls, the template arguments that can be deduced from the function arguments can be omitted (§13.3.1). For example:

```
template int convert<int,double>(double);      // ok (redundant)
template int convert<int>(double);             // ok
```

When a class template is explicitly instantiated, every member function is also instantiated.

Note that an explicit instantiation can be used as a constraints check (§13.6.2). For example:

```
template<class T> class Calls_foo {
    void constraints(T t) { foo(t); }      // call from every constructor
    // ...
};

template class Calls_foo<int>;                 // error: foo(int) undefined
template Calls_foo<Shape*>::constraints();     // error: foo(Shape*) undefined
```

The link-time and recompilation efficiency impact of instantiation requests can be significant. I have seen examples in which bundling most template instantiations into a single compilation unit cut the compile time from a number of hours to the equivalent number of minutes.

It is an error to have two definitions for the same specialization. It does not matter if such multiple specializations are user-defined (§13.5), implicitly generated (§C.13.7), or explicitly requested. However, a compiler is not required to diagnose multiple instantiations in separate compilation units. This allows a smart implementation to ignore redundant instantiations and thereby avoid problems related to composition of programs from libraries using explicit instantiation (§C.13.7). However, implementations are not required to be smart. Users of ''less smart'' implementations must avoid multiple instantiations. However, the worst that will happen if they don't is that their program won't load; there will be no silent changes of meaning.

The language does not require that a user request explicit instantiation. Explicit instantiation is an optional mechanism for optimization and manual control of the compile-and-link process (§C.13.7).

C.14 Advice

[1] Focus on software development rather than technicalities; §C.1.
[2] Adherence to the standard does not guarantee portability; §C.2.
[3] Avoid undefined behavior (including proprietary extensions); §C.2.
[4] Localize implementation-defined behavior; §C.2.
[5] Use keywords and digraphs to represent programs on systems where `{ } [ ] |` are missing and trigraphs if `\` or `!` are missing; §C.3.1.
[6] To ease communication, use the ANSI characters to represent programs; §C.3.3.
[7] Prefer symbolic escape characters to numeric representation of characters; §C.3.2.
[8] Do not rely on signedness or unsignedness of ***char***; §C.3.4.
[9] If in doubt about the type of an integer literal, use a suffix; §C.4.
[10] Avoid value-destroying implicit conversions; §C.6.
[11] Prefer ***vector*** over array; §C.7.
[12] Avoid ***union***s; §C.8.2.

[13] Use fields to represent externally-imposed layouts; §C.8.1.
[14] Be aware of the tradeoffs between different styles of memory management; §C.9.
[15] Don't pollute the global namespace; §C.10.1.
[16] Where a scope (module) rather than a type is needed, prefer a ***namespace*** over a ***class***; §C.10.3.
[17] Remember to define ***static*** class template members; §C.13.1.
[18] Use ***typename*** to disambiguate type members of a template parameter; §C.13.5.
[19] Where explicit qualification by template arguments is necessary, use ***template*** to disambiguate template class members; §C.13.6.
[20] Write template definitions with minimal dependence on their instantiation context; §C.13.8.
[21] If template instantiation takes too long, consider explicit instantiation; §C.13.10.
[22] If the order of compilation needs to be perfectly predictable, consider explicit instantiation; §C.13.10.