7

Pointers, Arrays, and References

The sublime and the ridiculous
are often so nearly related that
it is difficult to class them separately.
– Thomas Paine

7.1 Introduction

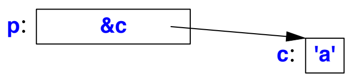
This chapter deals with the basic language mechanisms for referring to memory. Obviously, we can refer to an object by name, but in C++ (most) objects "have identity." That is, they reside at a specific address in memory, and an object can be accessed if you know its address and its type. The language constructs for holding and using addresses are pointers and references.

7.2 Pointers

For a type T, T* is the type “pointer to T.” That is, a variable of type T* can hold the address of an object of type T. For example:

```
char c = 'a';
char* p = &c;          // p holds the address of c; & is the address-of operator
```

or graphically:



The fundamental operation on a pointer is *dereferencing*, that is, referring to the object pointed to by the pointer. This operation is also called *indirection*. The dereferencing operator is (prefix) unary *. For example:

```
char c = 'a';
char* p = &c;   // p holds the address of c; & is the address-of operator
char c2 = *p;   // c2 == 'a'; * is the dereference operator
```

The object pointed to by p is c, and the value stored in c is 'a', so the value of *p assigned to c2 is 'a'.

It is possible to perform some arithmetic operations on pointers to array elements (§7.4).

The implementation of pointers is intended to map directly to the addressing mechanisms of the machine on which the program runs. Most machines can address a byte. Those that can’t tend to have hardware to extract bytes from words. On the other hand, few machines can directly address an individual bit. Consequently, the smallest object that can be independently allocated and pointed to using a built-in pointer type is a char. Note that a bool occupies at least as much space as a char (§6.2.8). To store smaller values more compactly, you can use the bitwise logical operations (§11.1.1), bit-fields in structures (§8.2.7), or a bitset (§34.2.2).

The *, meaning “pointer to,” is used as a suffix for a type name. Unfortunately, pointers to arrays and pointers to functions need a more complicated notation:

```
int* pi;               // pointer to int
char** ppc;            // pointer to pointer to char
int* ap[15];           // array of 15 pointers to ints
int (*fp)(char*);      // pointer to function taking a char* argument; returns an int
int* f(char*);         // function taking a char* argument; returns a pointer to int
```

See §6.3.1 for an explanation of the declaration syntax and §iso.A for the complete grammar.

Pointers to functions can be useful; they are discussed in §12.5. Pointers to class members are presented in §20.6.

7.2.1 void*

In low-level code, we occasionally need to store or pass along an address of a memory location without actually knowing what type of object is stored there. A void* is used for that. You can read void* as “pointer to an object of unknown type.”

A pointer to any type of object can be assigned to a variable of type void*, but a pointer to function (§12.5) or a pointer to member (§20.6) cannot. In addition, a void* can be assigned to another void*, void*s can be compared for equality and inequality, and a void* can be explicitly converted to another type. Other operations would be unsafe because the compiler cannot know what kind of object is really pointed to. Consequently, other operations result in compile-time errors. To use a void*, we must explicitly convert it to a pointer to a specific type. For example:

```
void f(int* pi)
{
        void* pv = pi;  // ok: implicit conversion of int* to void*
        *pv;            // error: can't dereference void*
        ++pv;           // error: can't increment void* (the size of the object pointed to is unknown)

        int* pi2 = static_cast<int*>(pv);                // explicit conversion back to int*

        double* pd1 = pv;                                // error
        double* pd2 = pi;                                // error
        double* pd3 = static_cast<double*>(pv);          // unsafe (§11.5.2)
}
```

In general, it is not safe to use a pointer that has been converted (''cast'') to a type that differs from the type of the object pointed to. For example, a machine may assume that every double is allocated on an 8-byte boundary. If so, strange behavior could arise if pi pointed to an int that wasn't allocated that way. This form of explicit type conversion is inherently unsafe and ugly. Consequently, the notation used, static_cast (§11.5.2), was designed to be ugly and easy to find in code.

The primary use for void* is for passing pointers to functions that are not allowed to make assumptions about the type of the object and for returning untyped objects from functions. To use such an object, we must use explicit type conversion.

Functions using void* pointers typically exist at the very lowest level of the system, where real hardware resources are manipulated. For example:

```
void* my_alloc(size_t n);       // allocate n bytes from my special heap
```

Occurrences of void*s at higher levels of the system should be viewed with great suspicion because they are likely indicators of design errors. Where used for optimization, void* can be hidden behind a type-safe interface (§27.3.1).

Pointers to functions (§12.5) and pointers to members (§20.6) cannot be assigned to void*s.

7.2.2 nullptr

The literal nullptr represents the null pointer, that is, a pointer that does not point to an object. It can be assigned to any pointer type, but not to other built-in types:

```
int* pi = nullptr;
double* pd = nullptr;
int i = nullptr;                // error: i is not a pointer
```

There is just one nullptr, which can be used for every pointer type, rather than a null pointer for each pointer type.

Before **nullptr** was introduced, zero (**0**) was used as a notation for the null pointer. For example:

```
int* x = 0; // x gets the value nullptr
```

No object is allocated with the address **0**, and **0** (the all-zeros bit pattern) is the most common representation of **nullptr**. Zero (**0**) is an **int**. However, the standard conversions (§10.5.2.3) allow **0** to be used as a constant of pointer or pointer-to-member type.

It has been popular to define a macro **NULL** to represent the null pointer. For example:

```
int* p = NULL; // using the macro NULL
```

However, there are differences in the definition of **NULL** in different implementations; for example, **NULL** might be **0** or **0L**. In C, **NULL** is typically **(void*)0**, which makes it illegal in C++ (§7.2.1):

```
int* p = NULL; // error: can't assign a void* to an int*
```

Using **nullptr** makes code more readable than alternatives and avoids potential confusion when a function is overloaded to accept either a pointer or an integer (§12.3.1).

7.3 Arrays

For a type **T**, **T[size]** is the type "array of **size** elements of type **T**." The elements are indexed from **0** to **size−1**. For example:

```
float v[3];      // an array of three floats: v[0], v[1], v[2]
char* a[32];     // an array of 32 pointers to char: a[0] .. a[31]
```

You can access an array using the subscript operator, **[]**, or through a pointer (using operator * or operator **[]**; §7.4). For example:

```
void f()
{
     int aa[10];
     aa[6] = 9;       // assign to aa's 7th element
     int x = aa[99];  // undefined behavior
}
```

Access out of the range of an array is undefined and usually disastrous. In particular, run-time range checking is neither guaranteed nor common.

The number of elements of the array, the array bound, must be a constant expression (§10.4). If you need variable bounds, use a **vector** (§4.4.1, §31.4). For example:

```
void f(int n)
{
     int v1[n];            // error: array size not a constant expression
     vector<int> v2(n);    // OK: vector with n int elements
}
```

Multidimensional arrays are represented as arrays of arrays (§7.4.2).

An array is C++'s fundamental way of representing a sequence of objects in memory. If what you want is a simple fixed-length sequence of objects of a given type in memory, an array is the ideal solution. For every other need, an array has serious problems.

An array can be allocated statically, on the stack, and on the free store (§6.4.2). For example:

```
int a1[10];                    // 10 ints in static storage

void f()
{
      int a2 [20];             // 20 ints on the stack
      int*p = new int[40];     // 40 ints on the free store
      // ...
}
```

The C++ built-in array is an inherently low-level facility that should primarily be used inside the implementation of higher-level, better-behaved, data structures, such as the standard-library **vector** or **array**. There is no array assignment, and the name of an array implicitly converts to a pointer to its first element at the slightest provocation (§7.4). In particular, avoid arrays in interfaces (e.g., as function arguments; §7.4.3, §12.2.2) because the implicit conversion to pointer is the root cause of many common errors in C code and C-style C++ code. If you allocate an array on the free store, be sure to **delete[]** its pointer once only and only after its last use (§11.2.2). That's most easily and most reliably done by having the lifetime of the free-store array controlled by a resource handle (e.g., **string** (§19.3, §36.3), **vector** (§13.6, §34.2), or **unique_ptr** (§34.3.1)). If you allocate an array statically or on the stack, be sure never to **delete[]** it. Obviously, C programmers cannot follow these pieces of advice because C lacks the ability to encapsulate arrays, but that doesn't make the advice bad in the context of C++.

One of the most widely used kinds of arrays is a zero-terminated array of **char**. That's the way C stores strings, so a zero-terminated array of **char** is often called a *C-style string*. C++ string literals follow that convention (§7.3.2), and some standard-library functions (e.g., **strcpy()** and **strcmp()**; §43.4) rely on it. Often, a **char*** or a **const char*** is assumed to point to a zero-terminated sequence of characters.

7.3.1 Array Initializers

An array can be initialized by a list of values. For example:

```
int v1[] = { 1, 2, 3, 4 };
char v2[] = { 'a', 'b', 'c', 0 };
```

When an array is declared without a specific size, but with an initializer list, the size is calculated by counting the elements of the initializer list. Consequently, **v1** and **v2** are of type **int[4]** and **char[4]**, respectively. If a size is explicitly specified, it is an error to give surplus elements in an initializer list. For example:

```
char v3[2] = { 'a', 'b', 0 };     // error: too many initializers
char v4[3] = { 'a', 'b', 0 };     // OK
```

If the initializer supplies too few elements for an array, **0** is used for the rest. For example:

```
int v5[8] = { 1, 2, 3, 4 };
```

is equivalent to

```
int v5[] = { 1, 2, 3, 4 , 0, 0, 0, 0 };
```

There is no built-in copy operation for arrays. You cannot initialize one array with another (not even of exactly the same type), and there is no array assignment:

```
int v6[8] = v5;  // error: can't copy an array (cannot assign an int* to an array)
v6 = v5;         // error: no array assignment
```

Similarly, you can't pass arrays by value. See also §7.4.

When you need assignment to a collection of objects, use a **vector** (§4.4.1, §13.6, §34.2), an **array** (§8.2.4), or a **valarray** (§40.5) instead.

An array of characters can be conveniently initialized by a string literal (§7.3.2).

7.3.2 String Literals

A *string literal* is a character sequence enclosed within double quotes:

```
"this is a string"
```

A string literal contains one more character than it appears to have; it is terminated by the null character, **'\0'**, with the value **0**. For example:

```
sizeof("Bohr")==5
```

The type of a string literal is "array of the appropriate number of **const** characters," so **"Bohr"** is of type **const char[5]**.

In C and in older C++ code, you could assign a string literal to a non-**const char***:

```
void f()
{
    char* p = "Plato";   // error, but accepted in pre-C++11-standard code
    p[4] = 'e';          // error: assignment to const
}
```

It would obviously be unsafe to accept that assignment. It was (and is) a source of subtle errors, so please don't grumble too much if some old code fails to compile for this reason. Having string literals immutable is not only obvious but also allows implementations to do significant optimizations in the way string literals are stored and accessed.

If we want a string that we are guaranteed to be able to modify, we must place the characters in a non-**const** array:

```
void f()
{
    char p[] = "Zeno";   // p is an array of 5 char
    p[0] = 'R';          // OK
}
```

A string literal is statically allocated so that it is safe to return one from a function. For example:

```
const char* error_message(int i)
{
    // ...
    return "range error";
}
```

The memory holding "range error" will not go away after a call of error_message().

Whether two identical string literals are allocated as one array or as two is implementation-defined (§6.1). For example:

```
const char* p = "Heraclitus";
const char* q = "Heraclitus";

void g()
{
        if (p == q) cout << "one!\n";          // the result is implementation-defined
        // ...
}
```

Note that == compares addresses (pointer values) when applied to pointers, and not the values pointed to.

The empty string is written as a pair of adjacent double quotes, "", and has the type const char[1]. The one character of the empty string is the terminating '\0'.

The backslash convention for representing nongraphic characters (§6.2.3.2) can also be used within a string. This makes it possible to represent the double quote (") and the escape character backslash (\) within a string. The most common such character by far is the newline character, '\n'. For example:

```
cout<<"beep at end of message\a\n";
```

The escape character, '\a', is the ASCII character BEL (also known as *alert*), which causes a sound to be emitted.

It is not possible to have a “real” newline in a (nonraw) string literal:

```
"this is not a string
but a syntax error"
```

Long strings can be broken by whitespace to make the program text neater. For example:

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz"
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

The compiler will concatenate adjacent strings, so alpha could equivalently have been initialized by the single string

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

It is possible to have the null character in a string, but most programs will not suspect that there are characters after it. For example, the string "Jens\000Munk" will be treated as "Jens" by standard-library functions such as strcpy() and strlen(); see §43.4.

7.3.2.1 Raw Character Strings

To represent a backslash (\) or a double quote (") in a string literal, we have to precede it with a backslash. That’s logical and in most cases quite simple. However, if we need a lot of backslashes and a lot of quotes in string literals, this simple technique becomes unmanageable. In particular, in regular expressions a backslash is used both as an escape character and to introduce characters representing character classes (§37.1.1). This is a convention shared by many programming languages, so we can't just change it. Therefore, when you write regular expressions for use with the standard **regex** library (Chapter 37), the fact that a backslash is an escape character becomes a notable source of errors. Consider how to write the pattern representing two words separated by a backslash (\):

```
string s = "\\w\\\\w";          // I hope I got that right
```

To prevent the frustration and errors caused by this clash of conventions, C++ provides *raw string literals*. A raw string literal is a string literal where a backslash is just a backslash (and a double quote is just a double quote) so that our example becomes:

```
string s = R"(\w\\w)";          // I'm pretty sure I got that right
```

Raw string literals use the **R"(ccc)"** notation for a sequence of characters **ccc**. The initial **R** is there to distinguish raw string literals from ordinary string literals. The parentheses are there to allow ("unescaped") double quotes. For example:

```
R"("quoted string")"            // the string is "quoted string"
```

So, how do we get the character sequence **)"** into a raw string literal? Fortunately, that's a rare problem, but **"(** and **)"** is only the default delimiter pair. We can add delimiters before the **(** and after the **)** in **"(...)"**. For example:

```
R"***("quoted string containing the usual terminator ("))")***"
        // "quoted string containing the usual terminator ("))"
```

The character sequence after the **)** must be identical to the sequence before the **(**. This way we can cope with (almost) arbitrarily complicated patterns.

Unless you work with regular expressions, raw string literals are probably just a curiosity (and one more thing to learn), but regular expressions are useful and widely used. Consider a real-world example:

```
"('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"          // Are the five backslashes correct or not?
```

With examples like that, even experts easily become confused, and raw string literals provide a significant service.

In contrast to nonraw string literals, a raw string literal can contain a newline. For example:

```
string counts {R"(1
22
333)"};
```

is equivalent to

```
string x {"1\n22\n333"};
```

7.3.2.2 Larger Character Sets

A string with the prefix **L**, such as **L"angst"**, is a string of wide characters (§6.2.3). Its type is **const wchar_t[]**. Similarly, a string with the prefix **LR**, such as **LR"(angst)"**, is a raw string (§7.3.2.1) of wide characters of type **const wchar_t[]**. Such a string is terminated by a **L'\0'** character.

There are six kinds of character literals supporting Unicode (*Unicode literals*). This sounds excessive, but there are three major encodings of Unicode: UTF-8, UTF-16, and UTF-32. For each of these three alternatives, both raw and ‘‘ordinary’’ strings are supported. All three UTF encodings support all Unicode characters, so which you use depends on the system you need to fit into. Essentially all Internet applications (e.g., browsers and email) rely on one or more of these encodings.

UTF-8 is a variable-width encoding: common characters fit into 1 byte, less frequently used characters (by some estimate of use) into 2 bytes, and rarer characters into 3 or 4 bytes. In particular, the ASCII characters fit into 1 byte with the same encodings (integer values) in UTF-8 as in ASCII. The various Latin alphabets, Greek, Cyrillic, Hebrew, Arabic, and more fit into 2 bytes.

A UTF-8 string is terminated by `'\0'`, a UTF-16 string by `u'\0'`, and a UTF-32 string by `U'\0'`.

We can represent an ordinary English character string in a variety of ways. Consider a file name using a backslash as the separator:

```
"folder\\file"       // implementation character set string
R"(folder\file)"     // implementation character raw set string
u8"folder\\file"     // UTF-8 string
u8R"(folder\file)"   // UTF-8 raw string
u"folder\\file"      // UTF-16 string
uR"(folder\file)"        // UTF-16 raw string
U"folder\\file"      // UTF-32 string
UR"(folder\file)"        // UTF-32 raw string
```

If printed, these strings will all look the same, but except for the ‘‘plain’’ and UTF-8 strings their internal representations are likely to differ.

Obviously, the real purpose of Unicode strings is to be able to put Unicode characters into them. For example:

```
u8"The official vowels in Danish are: a, e, i, o, u, \u00E6, \u00F8, \u00E5 and y."
```

Printing that string appropriately gives you

```
The official vowels in Danish are: a, e, i, o, u, æ, ø, å and y.
```

The hexadecimal number after the `\u` is a Unicode code point (§iso.2.14.3) [Unicode,1996]. Such a code point is independent of the encoding used and will in fact have different representations (as bits in bytes) in different encodings. For example, `u'0430'` (Cyrillic lowercase letter ‘‘a’’) is the 2-byte hexadecimal value `D0B0` in UTF-8, the 2-byte hexadecimal value `0403` in UTF-16, and the 4-byte hexadecimal value `00000403` in UTF-32. These hexadecimal values are referred to as *universal character names*.

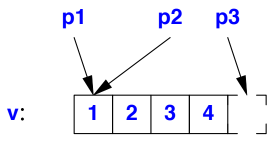
The order of the `u`s and `R`s and their cases are significant: `RU` and `Ur` are not valid string prefixes.

7.4 Pointers into Arrays

In C++, pointers and arrays are closely related. The name of an array can be used as a pointer to its initial element. For example:

```
int v[] = { 1, 2, 3, 4 };
int∗ p1 = v;          // pointer to initial element (implicit conversion)
int∗ p2 = &v[0];      // pointer to initial element
int∗ p3 = v+4;        // pointer to one-beyond-last element
```

or graphically:



Taking a pointer to the element one beyond the end of an array is guaranteed to work. This is important for many algorithms (§4.5, §33.1). However, since such a pointer does not in fact point to an element of the array, it may not be used for reading or writing. The result of taking the address of the element before the initial element or beyond one-past-the-last element is undefined and should be avoided. For example:

```
int∗ p4 = v−1;   // before the beginning, undefined: don't do it
int∗ p5 = v+7;   // beyond the end, undefined: don't do it
```

The implicit conversion of an array name to a pointer to the initial element of the array is extensively used in function calls in C-style code. For example:

```
extern "C" int strlen(const char∗);       // from <string.h>

void f()
{
      char v[] = "Annemarie";
      char∗ p = v;      // implicit conversion of char[] to char*
      strlen(p);
      strlen(v);        // implicit conversion of char[] to char*
      v = p;            // error: cannot assign to array
}
```

The same value is passed to the standard-library function **strlen()** in both calls. The snag is that it is impossible to avoid the implicit conversion. In other words, there is no way of declaring a function so that the array **v** is copied when the function is called. Fortunately, there is no implicit or explicit conversion from a pointer to an array.

The implicit conversion of the array argument to a pointer means that the size of the array is lost to the called function. However, the called function must somehow determine the size to perform a meaningful operation. Like other C standard-library functions taking pointers to characters, **strlen()** relies on zero to indicate end-of-string; **strlen(p)** returns the number of characters up to and not including the terminating **0**. This is all pretty low-level. The standard-library **vector** (§4.4.1, §13.6, §31.4), **array** (§8.2.4, §34.2.1), and **string** (§4.2) don't suffer from this problem. These library types give their number of elements as their **size()** without having to count elements each time.

7.4.1 Navigating Arrays

Efficient and elegant access to arrays (and similar data structures) is the key to many algorithms (see §4.5, Chapter 32). Access can be achieved either through a pointer to an array plus an index or through a pointer to an element. For example:

```
void fi(char v[])
{
      for (int i = 0; v[i]!=0; ++i)
            use(v[i]);
}

void fp(char v[])
{
      for (char* p = v; *p!=0; ++p)
            use(*p);
}
```

The prefix * operator dereferences a pointer so that *p is the character pointed to by p, and ++ increments the pointer so that it refers to the next element of the array.

There is no inherent reason why one version should be faster than the other. With modern compilers, identical code should be (and usually is) generated for both examples. Programmers can choose between the versions on logical and aesthetic grounds.

Subscripting a built-in array is defined in terms of the pointer operations + and *. For every built-in array a and integer j within the range of a, we have:

```
a[j] == *(&a[0]+j) == *(a+j) == *(j+a) == j[a]
```

It usually surprises people to find that a[j]==j[a]. For example, 3["Texas"]=="Texas"[3]=='a'. Such cleverness has no place in production code. These equivalences are pretty low-level and do not hold for standard-library containers, such as array and vector.

The result of applying the arithmetic operators +, -, ++, or -- to pointers depends on the type of the object pointed to. When an arithmetic operator is applied to a pointer p of type T*, p is assumed to point to an element of an array of objects of type T; p+1 points to the next element of that array, and p-1 points to the previous element. This implies that the integer value of p+1 will be sizeof(T) larger than the integer value of p. For example:

```
template<typename T>
int byte_diff(T* p, T* q)
{
      return reinterpret_cast<char*>(q)-reinterpret_cast<char*>(p);
}

void diff_test()
{
      int vi[10];
      short vs[10];
```

```
        cout << vi << ' ' << &vi[1] << ' ' << &vi[1]−&vi[0] << ' ' << byte_diff(&vi[0],&vi[1]) << '\n';
        cout << vs << ' ' << &vs[1] << ' ' << &vs[1]−&vs[0] << ' ' << byte_diff(&vs[0],&vs[1]) << '\n';
}
```

This produced:

```
0x7fffaef0 0x7fffaef4 1 4
0x7fffaedc 0x7fffaede 1 2
```

The pointer values were printed using the default hexadecimal notation. This shows that on my implementation, `sizeof(short)` is `2` and `sizeof(int)` is `4`.

Subtraction of pointers is defined only when both pointers point to elements of the same array (although the language has no fast way of ensuring that is the case). When subtracting a pointer `p` from another pointer `q`, `q−p`, the result is the number of array elements in the sequence [`p`:`q`) (an integer). One can add an integer to a pointer or subtract an integer from a pointer; in both cases, the result is a pointer value. If that value does not point to an element of the same array as the original pointer or one beyond, the result of using that value is undefined. For example:

```
void f()
{
        int v1[10];
        int v2[10];

        int i1 = &v1[5]−&v1[3];      // i1 = 2
        int i2 = &v1[5]−&v2[3];      // result undefined

        int∗ p1 = v2+2;              // p1 = &v2[2]
        int∗ p2 = v2−2;              // *p2 undefined
}
```

Complicated pointer arithmetic is usually unnecessary and best avoided. Addition of pointers makes no sense and is not allowed.

Arrays are not self-describing because the number of elements of an array is not guaranteed to be stored with the array. This implies that to traverse an array that does not contain a terminator the way C-style strings do, we must somehow supply the number of elements. For example:

```
void fp(char v[], int size)
{
        for (int i=0; i!=size; ++i)
                use(v[i]);           // hope that v has at least size elements
        for (int x : v)
                use(x);              // error: range-for does not work for pointers

        const int N = 7;
        char v2[N];
        for (int i=0; i!=N; ++i)
                use(v2[i]);
        for (int x : v2)
                use(x);              // range-for works for arrays of known size
}
```

This array concept is inherently low-level. Most advantages of the built-in array and few of the disadvantages can be obtained through the use of the standard-library container **array** (§8.2.4, §34.2.1). Some C++ implementations offer optional range checking for arrays. However, such checking can be quite expensive, so it is often used only as a development aid (rather than being included in production code). If you are not using range checking for individual accesses, try to maintain a consistent policy of accessing elements only in well-defined ranges. That is best done when arrays are manipulated through the interface of a higher-level container type, such as **vector**, where it is harder to get confused about the range of valid elements.

7.4.2 Multidimensional Arrays

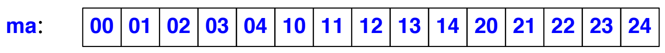
Multidimensional arrays are represented as arrays of arrays; a 3-by-5 array is declared like this:

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

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:



The array **ma** is simply 15 **int**s 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 (§10.3.2). 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];
```

7.4.3 Passing Arrays

Arrays cannot directly be passed by value. Instead, an array is passed as a pointer to its first element. For example:

```
void comp(double arg[10])          // arg is a double*
{
        for (int i=0; i!=10; ++i)
                arg[i]+=99;
}

void f()
{
        double a1[10];
        double a2[5];
        double a3[100];

        comp(a1);
        comp(a2);       // disaster!
        comp(a3);       // uses only the first 10 elements
};
```

This code looks sane, but it is not. The code compiles, but the call comp(a2) will write beyond the bounds of a2. Also, anyone who guessed that the array was passed by value will be disappointed: the writes to arg[i] are writes directly to the elements of comp()'s argument, rather than to a copy. The function could equivalently have been written as

```
void comp(double* arg)
{
        for (int i=0; i!=10; ++i)
                arg[i]+=99;
}
```

Now the insanity is (hopefully) obvious. When used as a function argument, the first dimension of an array is simply treated as a pointer. Any array bound specified is simply ignored. This implies that if you want to pass a sequence of elements without losing size information, you should not pass a built-in array. Instead, you can place the array inside a class as a member (as is done for std::array) or define a class that acts as a handle (as is done for std::string and std::vector).

If you insist on using arrays directly, you will have to deal with bugs and confusion without getting noticeable advantages in return. 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; §7.4). The first dimension of an array is irrelevant to 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 layout of ma above and note that by 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';
        }
}
```

When both dimensions need to be passed, the "obvious solution" 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';
        }
}
```

Fortunately, 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. However, 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 test()
{
        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 (§29.2.2, §40.5.2).

The standard `vector` (§31.4) doesn't suffer from these problems.

7.5 Pointers and const

C++ offers two related meanings of ''constant'':

- `constexpr`: Evaluate at compile time (§2.2.3, §10.4).
- `const`: Do not modify in this scope (§2.2.3).

Basically, `constexpr`'s role is to enable and ensure compile-time evaluation, whereas `const`'s primary role is to specify immutability in interfaces. This section is primarily concerned with the second role: interface specification.

Many objects don't have their values changed after initialization:

- Symbolic constants lead to more maintainable code than using literals directly in code.
- Many pointers are often read through but never written through.
- Most function parameters are read but not written to.

To express this notion of immutability after initialization, we can add `const` to the definition of an object. For example:

```
const int model = 90;                // model is a const
const int v[] = { 1, 2, 3, 4 };      // v[i] is a const
const int x;                         // error: no initializer
```

Because an object declared `const` cannot be assigned to, it must be initialized.

Declaring something `const` ensures that its value will not change within its scope:

```
void f()
{
        model = 200;    // error
        v[2] = 3;       // error
}
```

Note that `const` modifies a type; it restricts the ways in which an object can be used, rather than specifying how the constant is to be allocated. For example:

```
void g(const X* p)
{
        // can't modify *p here
}
```

```
void h()
{
        X val;      // val can be modified here
        g(&val);
        // ...
}
```

When using a pointer, two objects are involved: the pointer itself and the object pointed to. "Prefixing" a declaration of a pointer with **const** makes the object, but not the pointer, a constant. To declare a pointer itself, rather than the object pointed to, to be a constant, we use the declarator operator **∗const** instead of plain ∗. For example:

```
void f1(char∗ p)
{
        char s[] = "Gorm";

        const char∗ pc = s;             // pointer to constant
        pc[3] = 'g';                    // error: pc points to constant
        pc = p;                         // OK

        char ∗const cp = s;             // constant pointer
        cp[3] = 'a';                    // OK
        cp = p;                         // error: cp is constant

        const char ∗const cpc = s;      // const pointer to const
        cpc[3] = 'a';                   // error: cpc points to constant
        cpc = p;                        // error: cpc is constant
}
```

The declarator operator that makes a pointer constant is **∗const**. There is no **const∗** declarator operator, so a **const** appearing before the ∗ is taken to be part of the base type. For example:

```
char ∗const cp;         // const pointer to char
char const∗ pc;         // pointer to const char
const char∗ pc2;        // pointer to const char
```

Some people find it helpful to read such declarations right-to-left, for example, "**cp** is a **const** pointer to a **char**" and "**pc2** is a pointer to a **char const**."

An object that is a constant when accessed through one pointer may be variable when accessed in other ways. This is particularly useful for function arguments. By declaring a pointer argument **const**, the function is prohibited from modifying the object pointed to. For example:

```
const char∗ strchr(const char∗ p, char c);      // find first occurrence of c in p
char∗ strchr(char∗ p, char c);                  // find first occurrence of c in p
```

The first version is used for strings where the elements mustn't be modified and returns a pointer to **const** that does not allow modification. The second version is used for mutable strings.

You can assign the address of a non-**const** variable to a pointer to constant because no harm can come from that. However, the address of a constant cannot be assigned to an unrestricted pointer because this would allow the object's value to be changed. For example:

```
void f4()
{
    int a = 1;
    const int c = 2;
    const int* p1 = &c;  // OK
    const int* p2 = &a;  // OK
    int* p3 = &c;        // error: initialization of int* with const int*
    *p3 = 7;             // try to change the value of c
}
```

It is possible, but typically unwise, to explicitly remove the restrictions on a pointer to `const` by explicit type conversion (§16.2.9, §11.5).

7.6 Pointers and Ownership

A resource is something that has to be acquired and later released (§5.2). Memory acquired by `new` and released by `delete` (§11.2) and files opened by `fopen()` and closed by `fclose()` (§43.2) are examples of resources where the most direct handle to the resource is a pointer. This can be most confusing because a pointer is easily passed around in a program, and there is nothing in the type system that distinguishes a pointer that owns a resource from one that does not. Consider:

```
void confused(int* p)
{
    // delete p?
}

int global {7};

void f()
{
    X* pn = new int{7};
    int i {7};
    int q = &i;
    confused(pn);
    confused(q);
    confused(&global);
}
```

If `confused()` `delete`s `p` the program will seriously misbehave for the second two calls because we may not `delete` objects not allocated by `new` (§11.2). If `confused()` does not `delete` `p` the program leaks (§11.2.1). In this case, obviously `f()` must manage the lifetime of the object it creates on the free store, but in general keeping track of what needs to be `delete`d in a large program requires a simple and consistent strategy.

It is usually a good idea to immediately place a pointer that represents ownership in a resource handle class, such as `vector`, `string`, and `unique_ptr`. That way, we can assume that every pointer that is not within a resource handle is not an owner and must not be `delete`d. Chapter 13 discusses resource management in greater detail.

7.7 References

A pointer allows us to pass potentially large amounts of data around at low cost: instead of copying the data we simply pass its address as a pointer value. The type of the pointer determines what can be done to the data through the pointer. Using a pointer differs from using the name of an object in a few ways:

- We use a different syntax, for example, `*p` instead of `obj` and `p->m` rather than `obj.m`.
- We can make a pointer point to different objects at different times.
- We must be more careful when using pointers than when using an object directly: a pointer may be a `nullptr` or point to an object that wasn't the one we expected.

These differences can be annoying; for example, some programmers find `f(&x)` ugly compared to `f(x)`. Worse, managing pointer variables with varying values and protecting code against the possibility of `nullptr` can be a significant burden. Finally, when we want to overload an operator, say `+`, we want to write `x+y` rather than `&x+&y`. The language mechanism addressing these problems is called a *reference*. Like a pointer, a *reference* is an alias for an object, is usually implemented to hold a machine address of an object, and does not impose performance overhead compared to pointers, but it differs from a pointer in that:

- You access a reference with exactly the same syntax as the name of an object.
- A reference always refers to the object to which it was initialized.
- There is no "null reference," and we may assume that a reference refers to an object (§7.7.4).

A reference is an alternative name for an object, an alias. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators (Chapter 18) in particular. For example:

```
template<class T>
class vector {
        T* elem;
        // ...
public:
        T& operator[](int i) { return elem[i]; }                 // return reference to element
        const T& operator[](int i) const { return elem[i]; }     // return reference to const element

        void push_back(const T& a);                              // pass element to be added by reference
        // ...
};

void f(const vector<double>& v)
{
        double d1 = v[1];      // copy the value of the double referred to by v.operator[](1) into d1
        v[2] = 7;              // place 7 in the double referred to by the result of v.operator[](2)

        v.push_back(d1);       // give push_back() a reference to d1 to work with
}
```

The idea of passing function arguments by reference is as old as high-level programming languages (the first version of Fortran used that).

To reflect the lvalue/rvalue and const/non-const distinctions, there are three kinds of references:

- *lvalue references*: to refer to objects whose value we want to change
- const *references*: to refer to objects whose value we do not want to change (e.g., a constant)
- *rvalue references*: to refer to objects whose value we do not need to preserve after we have used it (e.g., a temporary)

Collectively, they are called *references*. The first two are both called *lvalue references*.

7.7.1 Lvalue References

In a type name, the notation X& means "reference to X." It is used for references to lvalues, so it is often called an *lvalue reference*. For example:

```
void f()
{
        int var = 1;
        int& r {var};       // r and var now refer to the same int
        int x = r;          // x becomes 1

        r = 2;              // var becomes 2
}
```

To ensure that a reference is a name for something (that is, that it is bound to an object), we must initialize the reference. For example:

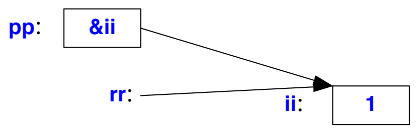
```
int var = 1;
int& r1 {var};          // OK: r1 initialized
int& r2;                // error: initializer missing
extern int& r3;         // OK: r3 initialized elsewhere
```

Initialization of a reference is something quite different from assignment to it. Despite appearances, no operator operates on a reference. For example:

```
void g()
{
        int var = 0;
        int& rr {var};
        ++rr;                   // var is incremented to 1
        int* pp = &rr;          // pp points to var
}
```

Here, ++rr does not increment the reference rr; rather, ++ is applied to the int to which rr refers, that is, to var. Consequently, the value of a reference cannot be changed after initialization; it always refers to the object it was initialized to denote. To get a pointer to the object denoted by a reference rr, we can write &rr. Thus, we cannot have a pointer to a reference. Furthermore, we cannot define an array of references. In that sense, a reference is not an object.

The obvious implementation of a reference is as a (constant) pointer that is dereferenced each time it is used. It doesn't do much harm to think about references that way, as long as one remembers that a reference isn't an object that can be manipulated the way a pointer is:



In some cases, the compiler can optimize away a reference so that there is no object representing that reference at run time.

Initialization of a reference is trivial when the initializer is an lvalue (an object whose address you can take; see §6.4). The initializer for a ‘‘plain’’ **T&** must be an lvalue of type **T**.

The initializer for a **const T&** need not be an lvalue or even of type **T**. In such cases:

[1] First, implicit type conversion to **T** is applied if necessary (see §10.5).
[2] Then, the resulting value is placed in a temporary variable of type **T**.
[3] Finally, this temporary variable is used as the value of the initializer.

Consider:

```
double& dr = 1;                // error: lvalue needed
const double& cdr {1};         // OK
```

The interpretation of this last initialization might be:

```
double temp = double{1};       // first create a temporary with the right value
const double& cdr {temp};      // then use the temporary as the initializer for cdr
```

A temporary created to hold a reference initializer persists until the end of its reference’s scope.

References to variables and references to constants are distinguished because introducing a temporary for a variable would have been highly error-prone; an assignment to the variable would become an assignment to the – soon-to-disappear – temporary. No such problem exists for references to constants, and references to constants are often important as function arguments (§18.2.4).

A reference can be used to specify a function argument so that the function can change the value of an object passed to it. For example:

```
void increment(int& aa)
{
        ++aa;
}

void f()
{
        int x = 1;
        increment(x);          // x = 2
}
```

The semantics of argument passing are defined to be those of initialization, so when called, **increment**’s argument **aa** became another name for **x**. To keep a program readable, it is often best to avoid functions that modify their arguments. Instead, you can return a value from the function explicitly:

```
int next(int p) { return p+1; }

void g()
{
        int x = 1;
        increment(x);           // x = 2
        x = next(x);            // x = 3
}
```

The increment(x) notation doesn't give a clue to the reader that x's value is being modified, the way x=next(x) does. Consequently, ''plain'' reference arguments should be used only where the name of the function gives a strong hint that the reference argument is modified.

References can also be used as return types. This is mostly used to define functions that can be used on both the left-hand and right-hand sides of an assignment. A Map is a good example. For example:

```
template<class K, class V>
class Map {             // a simple map class
public:
        V& operator[](const K& v);      // return the value corresponding to the key v

        pair<K,V>* begin() { return &elem[0]; }
        pair<K,V>* end() { return &elem[0]+elem.size(); }
private:
        vector<pair<K,V>> elem;         // {key,value} pairs
};
```

The standard-library map (§4.4.3, §31.4.3) is typically implemented as a red-black tree, but to avoid distracting implementation details, I'll just show an implementation based on linear search for a key match:

```
template<class K, class V>
V& Map<K,V>::operator[](const K& k)
{
        for (auto& x : elem)
                if (k == x.first)
                        return x.second;

        elem.push_back({k,V{}});        // add pair at end (§4.4.2)
        return elem.back().second;      // return the (default) value of the new element
}
```

I pass the key argument, k, by reference because it might be of a type that is expensive to copy. Similarly, I return the value by reference because it too might be of a type that is expensive to copy. I use a const reference for k because I don't want to modify it and because I might want to use a literal or a temporary object as an argument. I return the result by non-const reference because the user of a Map might very well want to modify the found value. For example:

```
int main() // count the number of occurrences of each word on input
{
        Map<string,int> buf;

        for (string s; cin>>s;) ++buf[s];

        for (const auto& x : buf)
                cout << x.first << ": " << x.second << '\n';
}
```

Each time around, the input loop reads one word from the standard input stream **cin** into the string **s** (§4.3.2) and then updates the counter associated with it. Finally, the resulting table of different words in the input, each with its number of occurrences, is printed. For example, given the input

```
aa bb bb aa aa bb aa aa
```

this program will produce

```
aa: 5
bb: 3
```

The range- **for** loop works for this because **Map** defined **begin()** and **end()**, just as is done for the standard-library **map**.

7.7.2 Rvalue References

The basic idea of having more than one kind of reference is to support different uses of objects:

- A non-**const** lvalue reference refers to an object, to which the user of the reference can write.
- A **const** lvalue reference refers to a constant, which is immutable from the point of view of the user of the reference.
- An rvalue reference refers to a temporary object, which the user of the reference can (and typically will) modify, assuming that the object will never be used again.

We want to know if a reference refers to a temporary, because if it does, we can sometimes turn an expensive copy operation into a cheap move operation (§3.3.2, §17.1, §17.5.2). An object (such as a **string** or a **list**) that is represented by a small descriptor pointing to a potentially huge amount of information can be simply and cheaply moved if we know that the source isn't going to be used again. The classic example is a return value where the compiler knows that a local variable returned will never again be used (§3.3.2).

An rvalue reference can bind to an rvalue, but not to an lvalue. In that, an rvalue reference is exactly opposite to an lvalue reference. For example:

```
string var {"Cambridge"};
string f();

string& r1 {var};               // lvalue reference, bind r1 to var (an lvalue)
string& r2 {f()};               // lvalue reference, error: f() is an rvalue
string& r3 {"Princeton"};       // lvalue reference, error: cannot bind to temporary
```

```
string&& rr1 {f()};            // rvalue reference, fine: bind rr1 to rvalue (a temporary)
string&& rr2 {var};            // rvalue reference, error: var is an lvalue
string&& rr3 {"Oxford"};       // rr3 refers to a temporary holding "Oxford"

const string cr1& {"Harvard"};// OK: make temporary and bind to cr1
```

The **&&** declarator operator means ''rvalue reference.'' We do *not* use **const** rvalue references; most of the benefits from using rvalue references involve writing to the object to which it refers. Both a **const** lvalue reference and an rvalue reference can bind to an rvalue. However, the purposes will be fundamentally different:

- We use rvalue references to implement a ''destructive read'' for optimization of what would otherwise have required a copy.
- We use a **const** lvalue reference to prevent modification of an argument.

An object referred to by an rvalue reference is accessed exactly like an object referred to by an lvalue reference or an ordinary variable name. For example:

```
string f(string&& s)
{
        if (s.size())
                s[0] = toupper(s[0]);
        return s;
}
```

Sometimes, a programmer knows that an object won't be used again, even though the compiler does not. Consider:

```
template<class T>
swap(T& a, T& b)           // "old-style swap"
{
        T tmp {a};// now we have two copies of a
        a = b;       // now we have two copies of b
        b = tmp;   // now we have two copies of tmp (aka a)
}
```

If **T** is a type for which it can be expensive to copy elements, such as **string** and **vector**, this **swap()** becomes an expensive operation. Note something curious: we didn't want any copies at all; we just wanted to move the values of **a**, **b**, and **tmp** around. We can tell that to the compiler:

```
template<class T>
void swap(T& a, T& b)      // "perfect swap" (almost)
{
        T tmp {static_cast<T&&>(a)}; // the initialization may write to a
        a = static_cast<T&&>(b);       // the assignment may write to b
        b = static_cast<T&&>(tmp);   // the assignment may write to tmp
}
```

The result value of **static_cast<T&&>(x)** is an rvalue of type **T&&** for **x**. An operation that is optimized for rvalues can now use its optimization for **x**. In particular, if a type **T** has a move constructor (§3.3.2, §17.5.2) or a move assignment, it will be used. Consider **vector**:

```
template<class T> class vector {
        // ...
        vector(const vector& r);  // copy constructor (copy r's representation)
        vector(vector&& r);        // move constructor ("steal" representation from r)
};

vector<string> s;
vector<string> s2 {s};                // s is an lvalue, so use copy constructor
vector<string> s3 {s+"tail"};         // s+"tail" is an rvalue so pick move constructor
```

The use of `static_cast` in `swap()` is a bit verbose and slightly prone to mistyping, so the standard library provides a `move()` function: `move(x)` means `static_cast<X&&>(x)` where `X` is the type of `x`. Given that, we can clean up the definition of `swap()` a bit:

```
template<class T>
void swap(T& a, T& b)     // "perfect swap" (almost)
{
        T tmp {move(a)};   // move from a
        a = move(b);       // move from b
        b = move(tmp);     // move from tmp
}
```

In contrast to the original `swap()`, this latest version need not make any copies; it will use move operations whenever possible.

Since `move(x)` does not move `x` (it simply produces an rvalue reference to `x`), it would have been better if `move()` had been called `rval()`, but by now `move()` has been used for years.

I deemed this `swap()` "almost perfect" because it will swap only lvalues. Consider:

```
void f(vector<int>& v)
{
        swap(v,vector<int>{1,2,3});     // replace v's elements with 1,2,3
        // ...
}
```

It is not uncommon to want to replace the contents of a container with some sort of default value, but this particular `swap()` cannot do that. A solution is to augment it by two overloads:

```
template<class T> void swap(T&& a, T& b);
template<class T> void swap(T& a, T&& b)
```

Our example will be handled by that last version of `swap()`. The standard library takes a different approach by defining `shrink_to_fit()` and `clear()` for `vector`, `string`, etc. (§31.3.3) to handle the most common cases of rvalue arguments to `swap()`:

```
void f(string& s, vector<int>& v)
{
        s.shrink_to_fit();         // make s.capacity()==s.size()
        swap(s,string{s});         // make s.capacity()==s.size()
        v.clear();                  // make v empty
        swap(v.vector<int>{});      // make v empty
        v = {};                     // make v empty
    }
```

Rvalue references can also be used to provide perfect forwarding (§23.5.2.1, §35.5.1).

All standard-library containers provide move constructors and move assignment (§31.3.2). Also, their operations that insert new elements, such as `insert()` and `push_back()`, have versions that take rvalue references.

7.7.3 References to References

It you take a reference to a reference to a type, you get a reference to that type, rather than some kind of special reference to reference type. But what kind of reference? Lvalue reference or rvalue reference? Consider:

```
using rr_i = int&&;
using lr_i = int&;
using rr_rr_i = rr_i&&;     // "int && &&" is an int&&
using lr_rr_i = rr_i&;      // "int && &" is an int&
using rr_lr_i = lr_i&&;     // "int & &&" is an int&
using lr_lr_i = lr_i&;      // "int & &" is an int&
```

In other words, lvalue reference always wins. This makes sense: nothing we can do with types can change the fact that an lvalue reference refers to an lvalue. This is sometimes known as *reference collapse*.

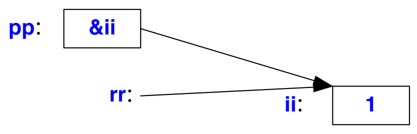
The syntax does not allow

```
int && & r = i;
```

Reference to reference can only happen as the result of an alias (§3.4.5, §6.5) or a template type argument (§23.5.2.1).

7.7.4 Pointers and References

Pointers and references are two mechanisms for referring to an object from different places in a program without copying. We can show this similarity graphically:



Each has its strengths and weaknesses.

If you need to change which object to refer to, use a pointer. You can use `=`, `+=`, `−=`, `++`, and `−−` to change the value of a pointer variable (§11.1.4). For example:

```
void fp(char∗ p)
{
        while (∗p)
                cout << ++∗p;
}

void fr(char& r)
{
        while (r)
                cout << ++r;    // oops: increments the char referred to, not the reference
                                // near-infinite loop!
}

void fr2(char& r)
{
        char∗ p = &r;           // get a pointer to the object referred to
        while (∗p)
                cout << ++∗p;
}
```

Conversely, if you want to be sure that a name always refers to the same object, use a reference. For example:

```
template<class T> class Proxy {         // Proxy refers to the object with which it is initialized
        T& m;
public:
        Proxy(T& mm) :m{mm} {}
        // ...
};

template<class T> class Handle {    // Handle refers to its current object
        T∗ m;
public:
        Proxy(T∗ mm) :m{mm} {}
        void rebind(T∗ mm) { m = mm; }
        // ...
};
```

If you want to use a user-defined (overloaded) operator (§18.1) on something that refers to an object, use a reference:

```
Matrix operator+(const Matrix&, const Matrix&);     // OK
Matrix operator−(const Matrix∗, const Matrix∗);     // error: no user-defined type argument

Matrix y, z;
// ...
Matrix x = y+z;         // OK
Matrix x2 = &y−&z;  // error and ugly
```

It is not possible to (re)define an operator for a pair of built-in types, such as pointers (§18.2.3).

If you want a collection of something that refers to an object, you must use a pointer:

```
int x, y;
string& a1[] = {x, y};              // error: array of references
string* a2[] = {&x, &y};            // OK
vector<string&> s1 = {x , y};       // error: vector of references
vector<string*> s2 = {&x, &y};      // OK
```

Once we leave the cases where C++ leaves no choice for the programmer, we enter the domain of aesthetics. Ideally, we will make our choices so as to minimize the probability of error and in particular to maximize readability of code.

If you need a notion of ‘‘no value,’’ pointers offer `nullptr`. There is no equivalent ‘‘null reference,’’ so if you need a ‘‘no value,’’ using a pointer may be most appropriate. For example:

```
void fp(X* p)
{
        if (p == nullptr) {
                // no value
        }
        else {
                // use *p
        }
}

void fr(X& r)      // common style
{
        // assume that r is valid and use it
}
```

If you really want to, you can construct and check for a ‘‘null reference’’ for a particular type:

```
void fr2(X& r)
{
        if (&r == &nullX) {     // or maybe r==nullX
                // no value
        }
        else {
                // use r
        }
}
```

Obviously, you need to have suitably defined `nullX`. The style is not idiomatic and I don’t recommend it. A programmer is allowed to assume that a reference is valid. It is possible to create an invalid reference, but you have to go out of your way to do so. For example:

```
char* ident(char * p) { return p; }

char& r {*ident(nullptr)}; // invalid code
```

This code is not valid C++ code. Don’t write such code even if your current implementation doesn’t catch it.

7.8 Advice

[1] Keep use of pointers simple and straightforward; §7.4.1.
[2] Avoid nontrivial pointer arithmetic; §7.4.
[3] Take care not to write beyond the bounds of an array; §7.4.1.
[4] Avoid multidimensional arrays; define suitable containers instead; §7.4.2.
[5] Use **nullptr** rather than **0** or **NULL**; §7.2.2.
[6] Use containers (e.g., **vector**, **array**, and **valarray**) rather than built-in (C-style) arrays; §7.4.1.
[7] Use **string** rather than zero-terminated arrays of **char**; §7.4.
[8] Use raw strings for string literals with complicated uses of backslash; §7.3.2.1.
[9] Prefer **const** reference arguments to plain reference arguments; §7.7.3.
[10] Use rvalue references (only) for forwarding and move semantics; §7.7.2.
[11] Keep pointers that represent ownership inside handle classes; §7.6.
[12] Avoid **void*** except in low-level code; §7.2.1.
[13] Use **const** pointers and **const** references to express immutability in interfaces; §7.5.
[14] Prefer references to pointers as arguments, except where "no object" is a reasonable option; §7.7.4.

This page intentionally left blank