5

Pointers, Arrays, and Structures

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

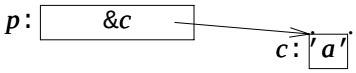
Pointers — zero — arrays — string literals — pointers into arrays — constants — pointers and constants — references — ***void**** — data structures — advice — exercises.

5.1 Pointers [ptr.ptr]

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
```

or graphically:



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 §4.9.1 for an explanation of the declaration syntax and Appendix A for the complete grammar.

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
char c2 = *p;   // c2 == 'a'
```

The variable 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 (§5.3). Pointers to functions can be extremely useful; they are discussed in §7.7.

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** (§4.6). To store smaller values more compactly, you can use logical operations (§6.2.4) or bit fields in structures (§C.8.1).

5.1.1 Zero [ptr.zero]

Zero (**0**) is an **int**. Because of standard conversions (§C.6.2.3), **0** can be used as a constant of any integral (§4.1.1), floating-point, pointer, or pointer-to-member type. The type of zero will be determined by context. Zero will typically (but not necessarily) be represented by the bit pattern *all-zeros* of the appropriate size.

No object is allocated with the address **0**. Consequently, **0** acts as a pointer literal, indicating that a pointer doesn't refer to an object.

In C, it has been popular to define a macro **NULL** to represent the zero pointer. Because of C++'s tighter type checking, the use of plain **0**, rather than any suggested **NULL** macro, leads to fewer problems. If you feel you must define **NULL**, use

```
const int NULL = 0;
```

The **const** qualifier (§5.4) prevents accidental redefinition of **NULL** and ensures that **NULL** can be used where a constant is required.

5.2 Arrays [ptr.array]

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]
```

The number of elements of the array, the array bound, must be a constant expression (§C.5). If you need variable bounds, use a ***vector*** (§3.7.1, §16.3). For example:

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

Multidimensional arrays are represented as arrays of arrays. For example:

```
int d2[10][20]; // d2 is an array of 10 arrays of 20 integers
```

Using comma notation as used for array bounds in some other languages gives compile-time errors because comma (`,`) is a sequencing operator (§6.2.2) and is not allowed in constant expressions (§C.5). For example, try this:

```
int bad[5,2];       // error: comma not allowed in a constant expression
```

Multidimensional arrays are described in §C.7. They are best avoided outside low-level code.

5.2.1 Array Initializers [ptr.array.init]

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, ***0*** is assumed for the remaining array elements. For example:

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

is equivalent to

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

Note that there is no array assignment to match the initialization:

```
void f()
{
    v4 = { 'c', 'd', 0 }; // error: no array assignment
}
```

When you need such assignments, use a ***vector*** (§16.3) or a ***valarray*** (§22.4) instead.

An array of characters can be conveniently initialized by a string literal (§5.2.2).

5.2.2 String Literals [ptr.string.literal]

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]**.

A string literal can be assigned to a **char***. This is allowed because in previous definitions of C and C++ , the type of a string literal was **char***. Allowing the assignment of a string literal to a **char*** ensures that millions of lines of C and C++ remain valid. It is, however, an error to try to modify a string literal through such a pointer:

```
void f()
{
    char* p = "Plato";
    p[4] = 'e';          // error: assignment to const; result is undefined
}
```

This kind of error cannot in general be caught until run-time, and implementations differ in their enforcement of this rule. Having string literals constant not only is 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 copy the characters into an 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 character literals are allocated as one is implementation-defined (§C.1). For example:

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

```
void g()
{
    if (p == q) cout << "one!\n";    // 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 backslash convention for representing nongraphic characters (§C.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 some kind of sound to be emitted.

It is not possible to have a ''real'' newline in a string:

```
"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 §20.4.1.

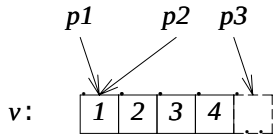
A string with the prefix *L*, such as *L"angst"*, is a string of wide characters (§4.3, §C.3.3). Its type is *const wchar_t[]*.

5.3 Pointers into Arrays [ptr.into]

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 (§2.7.2, §18.3). 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 is undefined and should be avoided. On some machine architectures, arrays are often allocated on machine addressing boundaries, so ''one before the initial element'' simply doesn't make sense.

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*** (§16.3) and ***string*** (Chapter 20) don't suffer from this problem.

5.3.1 Navigating Arrays [ptr.navigate]

Efficient and elegant access to arrays (and similar data structures) is the key to many algorithms (see §3.8, Chapter 18). Access can be achieved either through a pointer to an array plus an index or through a pointer to an element. For example, traversing a character string using an index,

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

is equivalent to a traversal using a pointer:

```
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 generated for both examples (see §5.9[8]). Programmers can choose between the versions on logical and aesthetic grounds.

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, executing

```
#include <iostream>

int main ()
{
    int vi[10];
    short vs[10];

    std::cout << &vi[0] << ' ' << &vi[1] << '\n';
    std::cout << &vs[0] << ' ' << &vs[1] << '\n';
}
```

produced

```
0x7fffaef0 0x7fffaef4
0x7fffaedc 0x7fffaede
```

using a default hexadecimal notation for pointer values. 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 one pointer from another, the result is the number of array elements between the two pointers (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 often 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 character strings do, we must somehow supply the number of elements. For example:

```
void fp(char v[], unsigned int size)
{
    for (int i=0; i<size; i++) use(v[i]);

    const int N = 7;
    char v2[N];
    for (int i=0; i<N; i++) use(v2[i]);

}
```

Note that most C++ implementations offer no range checking for arrays. This array concept is inherently low-level. A more advanced notion of arrays can be provided through the use of classes; see §3.7.1.

5.4 Constants [ptr.const]

C++ offers the concept of a user-defined constant, a ***const***, to express the notion that a value doesn't change directly. This is useful in several contexts. For example, many objects don't actually have their values changed after initialization, symbolic constants lead to more maintainable code than do literals embedded directly in code, pointers are often read through but never written through, and most function parameters are read but not written to.

The keyword ***const*** can be added to the declaration of an object to make the object declared a constant. Because it cannot be assigned to, a constant must be initialized. 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
```

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

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

Note that ***const*** modifies a type; that is, 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
    g(&val);
    // ...
}
```

Depending on how smart it is, a compiler can take advantage of an object being a constant in several ways. For example, the initializer for a constant is often (but not always) a constant expression (§C.5); if it is, it can be evaluated at compile time. Further, if the compiler knows every use of the ***const***, it need not allocate space to hold it. For example:

```
const int c1 = 1;
const int c2 = 2;
const int c3 = my_f(3);    // don't know the value of c3 at compile time
extern const int c4;       // don't know the value of c4 at compile time
const int* p = &c2;        // need to allocate space for c2
```

Given this, the compiler knows the values of *c1* and *c2* so that they can be used in constant expressions. Because the values of *c3* and *c4* are not known at compile time (using only the information available in this compilation unit; see §9.1), storage must be allocated for *c3* and *c4*. Because the address of *c2* is taken (and presumably used somewhere), storage must be allocated for *c2*. The simple and common case is the one in which the value of the constant is known at compile time and no storage needs to be allocated; *c1* is an example of that. The keyword ***extern*** indicates that *c4* is defined elsewhere (§9.2).

It is typically necessary to allocate store for an array of constants because the compiler cannot, in general, figure out which elements of the array are referred to in expressions. On many machines, however, efficiency improvements can be achieved even in this case by placing arrays of constants in read-only storage.

Common uses for ***const***s are as array bounds and case labels. For example:

```
const int a = 42;
const int b = 99;
const int max = 128;

int v[max];

void f(int i)
{
    switch (i) {
    case a:
        // ...
```

```
        case b:
            // ...
        }
    }
```

Enumerators (§4.8) are often an alternative to *const*s in such cases.

The way *const* can be used with class member functions is discussed in §10.2.6 and §10.2.7.

Symbolic constants should be used systematically to avoid ''magic numbers'' in code. If a numeric constant, such as an array bound, is repeated in code, it becomes hard to revise that code because every occurrence of that constant must be changed to make a correct update. Using a symbolic constant instead localizes information. Usually, a numeric constant represents an assumption about the program. For example, *4* may represent the number of bytes in an integer, *128* the number of characters needed to buffer input, and *6.24* the exchange factor between Danish kroner and U.S. dollars. Left as numeric constants in the code, these values are hard for a maintainer to spot and understand. Often, such numeric values go unnoticed and become errors when a program is ported or when some other change violates the assumptions they represent. Representing assumptions as well-commented symbolic constants minimizes such maintenance problems.

5.4.1 Pointers and Constants [ptr.pc]

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:

```
char* strcpy(char* p, const char* q); // cannot modify *q
```

You can assign the address of a 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 to explicitly remove the restrictions on a pointer to ***const*** by explicit type conversion (§10.2.7.1 and §15.4.2.1).

5.5 References [ptr.ref]

A *reference* is an alternative name for an object. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators (Chapter 11) in particular. The notation ***X***& means *reference to* ***X***. For example:

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

    r = 2;            // i = 2
}
```

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

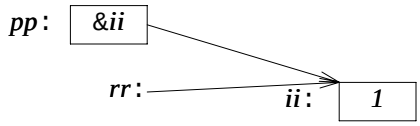
```
int i = 1;
int& r1 = i;          // ok: r1 initialized
int& r2;              // error: initializer missing
extern int& r3;       // ok: r3 initialized elsewhere
```

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 ii = 0;
    int& rr = ii;
    rr++;                   // ii is incremented to 1
    int* pp = &rr;          // pp points to ii
}
```

This is legal, but *rr*++ does not increment the reference *rr*; rather, ++ is applied to an *int* that happens to be *ii*. 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*.

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 thinking 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 §4.9.6). 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 §C.6);

[2] then, the resulting value is placed in a temporary variable of type *T*; and

[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 the introduction of a temporary in the case of the variable is 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 (§11.6).

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 or require a pointer argument:

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

void incr(int* p) { (*p)++; }

void g()
{
    int x = 1;
    increment(x);        // x = 2
    x = next(x);         // x = 3
    incr(&x);            // x = 4
}
```

The ***increment(x)*** notation doesn't give a clue to the reader that ***x***'s value is being modified, the way ***x=next(x)*** and ***incr(&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 to define functions that can be used on both the left-hand and right-hand sides of an assignment. Again, many of the most interesting uses of this are found in the design of nontrivial user-defined types. As an example, let us define a simple associative array. First, we define struct ***Pair*** like this:

```
struct Pair {
    string name;
    double val;
};
```

The basic idea is that a ***string*** has a floating-point value associated with it. It is easy to define a function, ***value()***, that maintains a data structure consisting of one ***Pair*** for each different string that has been presented to it. To shorten the presentation, a very simple (and inefficient) implementation is used:

```
vector<Pair> pairs;

double& value(const string& s)
/*
    maintain a set of Pairs:
    search for s, return its value if found; otherwise make a new Pair and return the default value 0
*/
{
```

```
        for (int i = 0; i < pairs.size(); i++)
                if (s == pairs[i].name) return pairs[i].val;

        Pair p = { s, 0 };
        pairs.push_back(p); // add Pair at end (§3.7.3)

        return pairs[pairs.size()-1].val;
}
```

This function can be understood as an array of floating-point values indexed by character strings. For a given argument string, ***value*()** finds the corresponding floating-point object (*not* the value of the corresponding floating-point object); it then returns a reference to it. For example:

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

        while (cin>>buf) value(buf)++;

        for (vector<Pair>::const_iterator p = pairs.begin(); p!=pairs.end(); ++p)
                cout << p->name << ": " << p->val << '\n';
}
```

Each time around, the ***while***-loop reads one word from the standard input stream ***cin*** into the string ***buf*** (§3.6) 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
```

It is easy to refine this into a proper associative array type by using a template class with the selection operator [] overloaded (§11.8). It is even easier just to use the standard library ***map*** (§17.4.1).

5.6 Pointer to Void [ptr.ptrtovoid]

A pointer of any type of object can be assigned to a variable of type ***void****, 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
}
```

In general, it is not safe to use a pointer that has been converted (''cast'') to a type that differs from the type 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***, was designed to be ugly.

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 suspicion because they are likely indicators of design errors. Where used for optimization, ***void**** can be hidden behind a type-safe interface (§13.5, §24.4.2).

Pointers to functions (§7.7) and pointers to members (§15.5) cannot be assigned to ***void****s.

5.7 Structures [ptr.struct]

An array is an aggregate of elements of the same type. A ***struct*** is an aggregate of elements of (nearly) arbitrary types. For example:

```
struct address {
    char* name;         // "Jim Dandy"
    long int number;    // 61
    char* street;       // "South St"
    char* town;         // "New Providence"
    char state[2];      // 'N' 'J'
    long zip;           // 7974
};
```

This defines a new type called ***address*** consisting of the items you need in order to send mail to someone. Note the semicolon at the end. This is one of very few places in C++ where it is necessary to have a semicolon after a curly brace, so people are prone to forget it.

Variables of type ***address*** can be declared exactly as other variables, and the individual *members* can be accessed using the `.` (dot) operator. For example:

```
void f()
{
    address jd;
    jd.name = "Jim Dandy";
    jd.number = 61;
}
```

The notation used for initializing arrays can also be used for initializing variables of structure types. For example:

```
address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence", {'N','J'}, 7974
};
```

Using a constructor (§10.2.3) is usually better, however. Note that `jd.state` could not be initialized by the string `"NJ"`. Strings are terminated by the character `'\0'`. Hence, `"NJ"` has three characters – one more than will fit into `jd.state`.

Structure objects are often accessed through pointers using the `->` (structure pointer dereference) operator. For example:

```
void print_addr(address* p)
{
    cout << p->name << '\n'
        << p->number << ' ' << p->street << '\n'
        << p->town << '\n'
        << p->state[0] << p->state[1] << ' ' << p->zip << '\n';
}
```

When `p` is a pointer, `p->m` is equivalent to `(*p).m`.

Objects of structure types can be assigned, passed as function arguments, and returned as the result from a function. For example:

```
address current;

address set_current(address next)
{
    address prev = current;
    current = next;
    return prev;
}
```

Other plausible operations, such as comparison (`==` and `!=`), are not defined. However, the user can define such operators (Chapter 11).

The size of an object of a structure type is not necessarily the sum of the sizes of its members. This is because many machines require objects of certain types to be allocated on architecture-dependent boundaries or handle such objects much more efficiently if they are. For example, integers are often allocated on word boundaries. On such machines, objects are said to have to be *aligned* properly. This leads to ''holes'' in the structures. For example, on many machines,

sizeof(*address*) is *24*, and not *22* as might be expected. You can minimize wasted space by simply ordering members by size (largest member first). However, it is usually best to order members for readability and sort them by size only if there is a demonstrated need to optimize.

The name of a type becomes available for use immediately after it has been encountered and not just after the complete declaration has been seen. For example:

```
struct Link {
    Link* previous;
    Link* successor;
};
```

It is not possible to declare new objects of a structure type until the complete declaration has been seen. For example:

```
struct No_good {
    No_good member;    // error: recursive definition
};
```

This is an error because the compiler is not able to determine the size of ***No_good***. To allow two (or more) structure types to refer to each other, we can declare a name to be the name of a structure type. For example:

```
struct List;      // to be defined later

struct Link {
    Link* pre;
    Link* suc;
    List* member_of;
};

struct List {
    Link* head;
};
```

Without the first declaration of ***List***, use of ***List*** in the declaration of ***Link*** would have caused a syntax error.

The name of a structure type can be used before the type is defined as long as that use does not require the name of a member or the size of the structure to be known. For example:

```
class S;   // 'S' is the name of some type

extern S a;
S f();
void g(S);
S* h(S*);
```

However, many such declarations cannot be used unless the type ***S*** is defined:

```
void k(S* p)
{
    S a;            // error: S not defined; size needed to allocate
```

```
    f();          // error: S not defined; size needed to return value
    g(a);         // error: S not defined; size needed to pass argument
    p->m = 7;     // error: S not defined; member name not known

    S* q = h(p);  // ok: pointers can be allocated and passed
    q->m = 7;     // error: S not defined; member name not known
}
```

A ***struct*** is a simple form of a ***class*** (Chapter 10).

For reasons that reach into the pre-history of C, it is possible to declare a ***struct*** and a non-structure with the same name in the same scope. For example:

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

In that case, the plain name (***stat***) is the name of the non-structure, and the structure must be referred to with the prefix ***struct***. Similarly, the keywords ***class***, ***union*** (§C.8.2), and ***enum*** (§4.8) can be used as prefixes for disambiguation. However, it is best not to overload names to make that necessary.

5.7.1 Type Equivalence [ptr.equiv]

Two structures are different types even when they have the same members. For example,

```
struct S1 { int a; };
struct S2 { int a; };
```

are two different types, so

```
S1 x;
S2 y = x;  // error: type mismatch
```

Structure types are also different from fundamental types, so

```
S1 x;
int i = x;  // error: type mismatch
```

Every ***struct*** must have a unique definition in a program (§9.2.3).

5.8 Advice [ptr.advice]

[1] Avoid nontrivial pointer arithmetic; §5.3.
[2] Take care not to write beyond the bounds of an array; §5.3.1.
[3] Use ***0*** rather than ***NULL***; §5.1.1.
[4] Use ***vector*** and ***valarray*** rather than built-in (C-style) arrays; §5.3.1.
[5] Use ***string*** rather than zero-terminated arrays of ***char***; §5.3.
[6] Minimize use of plain reference arguments; §5.5.
[7] Avoid ***void**** except in low-level code; §5.6.
[8] Avoid nontrivial literals (''magic numbers'') in code. Instead, define and use symbolic constants; §4.8, §5.4.

5.9 Exercises [ptr.exercises]

1. (∗1) Write declarations for the following: a pointer to a character, an array of 10 integers, a reference to an array of 10 integers, a pointer to an array of character strings, a pointer to a pointer to a character, a constant integer, a pointer to a constant integer, and a constant pointer to an integer. Initialize each one.
2. (∗1.5) What, on your system, are the restrictions on the pointer types ***char****, ***int****, and ***void****? For example, may an ***int**** have an odd value? Hint: alignment.
3. (∗1) Use ***typedef*** to define the types ***unsigned char***, ***const unsigned char***, pointer to integer, pointer to pointer to ***char***, pointer to arrays of ***char***, array of 7 pointers to ***int***, pointer to an array of 7 pointers to ***int***, and array of 8 arrays of 7 pointers to ***int***.
4. (∗1) Write a function that swaps (exchanges the values of) two integers. Use ***int**** as the argument type. Write another swap function using ***int&*** as the argument type.
5. (∗1.5) What is the size of the array ***str*** in the following example:

```
char str[] = "a short string";
```

   What is the length of the string ***"a short string"***?
6. (∗1) Define functions ***f(char)***, ***g(char&)***, and ***h(const char&)***. Call them with the arguments ***´a´***, ***49***, ***3300***, ***c***, ***uc***, and ***sc***, where ***c*** is a ***char***, ***uc*** is an ***unsigned char***, and ***sc*** is a ***signed char***. Which calls are legal? Which calls cause the compiler to introduce a temporary variable?
7. (∗1.5) Define a table of the names of months of the year and the number of days in each month. Write out that table. Do this twice; once using an array of ***char*** for the names and an array for the number of days and once using an array of structures, with each structure holding the name of a month and the number of days in it.
8. (∗2) Run some tests to see if your compiler really generates equivalent code for iteration using pointers and iteration using indexing (§5.3.1). If different degrees of optimization can be requested, see if and how that affects the quality of the generated code.
9. (∗1.5) Find an example where it would make sense to use a name in its own initializer.
10. (∗1) Define an array of strings in which the strings contain the names of the months. Print those strings. Pass the array to a function that prints those strings.
11. (∗2) Read a sequence of words from input. Use ***Quit*** as a word that terminates the input. Print the words in the order they were entered. Don't print a word twice. Modify the program to sort the words before printing them.
12. (∗2) Write a function that counts the number of occurrences of a pair of letters in a ***string*** and another that does the same in a zero-terminated array of ***char*** (a C-style string). For example, the pair "ab" appears twice in "xabaacbaxabb".
13. (∗1.5) Define a ***struct Date*** to keep track of dates. Provide functions that read ***Date***s from input, write ***Date***s to output, and initialize a ***Date*** with a date.

.