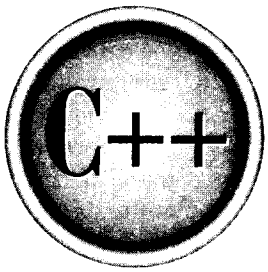
The Complete Reference



Chapter 5

Pointers

The correct understanding and use of pointers is critical to successful C/C++ programming. There are three reasons for this: First, pointers provide the means by which functions can modify their calling arguments. Second, pointers support dynamic allocation. Third, pointers can improve the efficiency of certain routines. Also, as you will see in Part Two, pointers take on additional roles in C++.

Pointers are one of the strongest but also one of the most dangerous features in C/C++. For example, uninitialized pointers (or pointers containing invalid values) can cause your system to crash. Perhaps worse, it is easy to use pointers incorrectly, causing bugs that are very difficult to find.

Because of both their importance and their potential for abuse, this chapter examines the subject of pointers in detail.

What Are Pointers?

A *pointer* is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to *point to* the second. Figure 5-1 illustrates this situation.

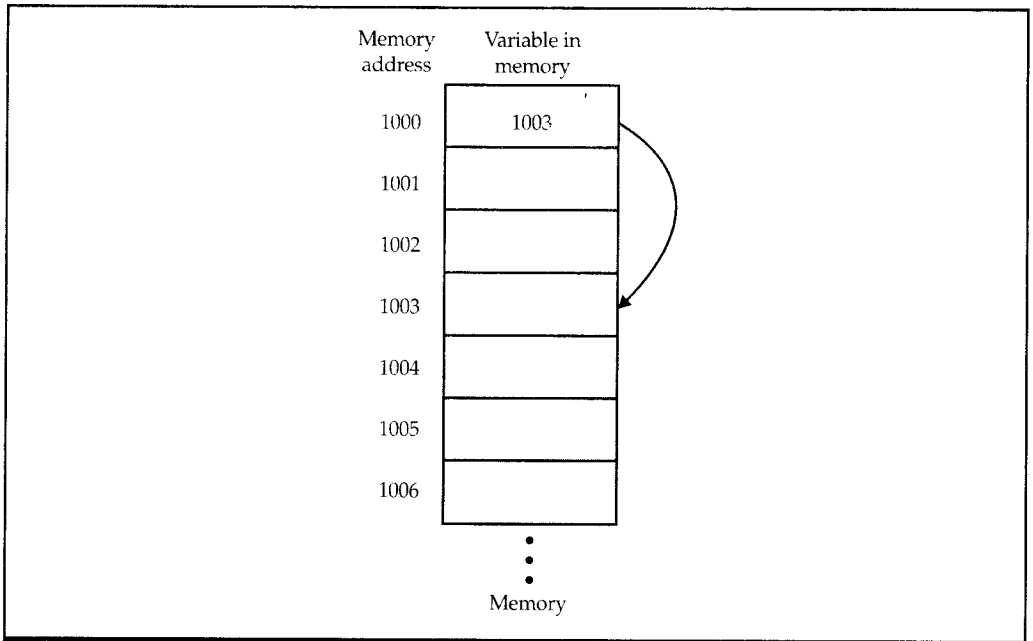


Figure 5-1. *One variable points to another*

Pointer Variables

If a variable is going to hold a pointer, it must be declared as such. A pointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is

type **name;*

where *type* is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by *name*.

The base type of the pointer defines what type of variables the pointer can point to. Technically, any type of pointer can point anywhere in memory. However, all pointer arithmetic is done relative to its base type, so it is important to declare the pointer correctly. (Pointer arithmetic is discussed later in this chapter.)

The Pointer Operators

The pointer operators were discussed in Chapter 2. We will take a closer look at them here, beginning with a review of their basic operation. There are two special pointer operators: * and &. The & is a unary operator that returns the memory address of its operand. (Remember, a unary operator only requires one operand.) For example,

```
m = &count;
```

places into **m** the memory address of the variable **count**. This address is the computer's internal location of the variable. It has nothing to do with the value of **count**. You can think of & as returning "the address of." Therefore, the preceding assignment statement means "**m** receives the address of **count**."

To understand the above assignment better, assume that the variable **count** uses memory location 2000 to store its value. Also assume that **count** has a value of 100. Then, after the preceding assignment, **m** will have the value 2000.

The second pointer operator, *, is the complement of &. It is a unary operator that returns the value located at the address that follows. For example, if **m** contains the memory address of the variable **count**,

```
q = *m;
```

places the value of **count** into **q**. Thus, **q** will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in **m**. You can think of * as "at address." In this case, the preceding statement means "**q** receives the value at address **m**."

Both & and * have a higher precedence than all other arithmetic operators except the unary minus, with which they are equal.

You must make sure that your pointer variables always point to the correct type of data. For example, when you declare a pointer to be of type **int**, the compiler assumes that any address that it holds points to an integer variable—whether it actually does or not. Because you can assign any address you want to a pointer variable, the following program compiles without error, but does not produce the desired result:

```
#include <stdio.h>

int main(void)
{
  double x = 100.1, y;
  int  *p;

  /* The next statement causes p (which is an
      integer pointer) to point to a double. */
  p = (int *)&x;

  /* The next statement does not operate as
      expected. */
  y = *p;


  printf("%f", y); /* won't output 100.1 */
  return 0;
}
```

This will not assign the value of **x** to **y**. Because **p** is declared as an integer pointer, only 4 bytes of information (assuming 4-byte integers) will be transferred to **y**, not the 8 bytes that normally make up a **double**.

Note *In C++, it is illegal to convert one type of pointer into another without the use of an explicit type cast. In C, casts should be used for most pointer conversions.*

Pointer Expressions

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions.

Pointer Assignments

As with any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. For example,

```
#include <stdio.h>

int main(void)
{
  int x;
  int *p1, *p2;

  p1 = &x;
  p2 = p1;

  printf(" %p", p2); /* print the address of x, not x's value! */

  return 0;
}
```

Both **p1** and **p2** now point to **x**. The address of **x** is displayed by using the **%p printf( )** format specifier, which causes **printf( )** to display an address in the format used by the host computer.

Pointer Arithmetic

There are only two arithmetic operations that you may use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let **p1** be an integer pointer with a current value of 2000. Also, assume integers are 2 bytes long. After the expression

```
p1++;
```

p1 contains 2002, not 2001. The reason for this is that each time **p1** is incremented, it will point to the next integer. The same is true of decrements. For example, assuming that **p1** has the value 2000, the expression

```
p1--;
```

causes **p1** to have the value 1998.

Generalizing from the preceding example, the following rules govern pointer arithmetic. Each time a pointer is incremented, it points to the memory location

of the next element of its base type. Each time it is decremented, it points to the location of the previous element. When applied to character pointers, this will appear as "normal" arithmetic because characters are always 1 byte long. All other pointers will increase or decrease by the length of the data type they point to. This approach ensures that a pointer is always pointing to an appropriate element of its base type. Figure 5-2 illustrates this concept.

You are not limited to the increment and decrement operators. For example, you may add or subtract integers to or from pointers. The expression

```
p1 = p1 + 12;
```

makes **p1** point to the twelfth element of **p1**'s type beyond the one it currently points to.

Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed: You may subtract one pointer from another in order to find the number of objects of their base type that separate the two. All other arithmetic operations are prohibited. Specifically, you may not multiply or divide pointers; you may not add two pointers; you may not apply the bitwise operators to them; and you may not add or subtract type **float** or **double** to or from pointers.

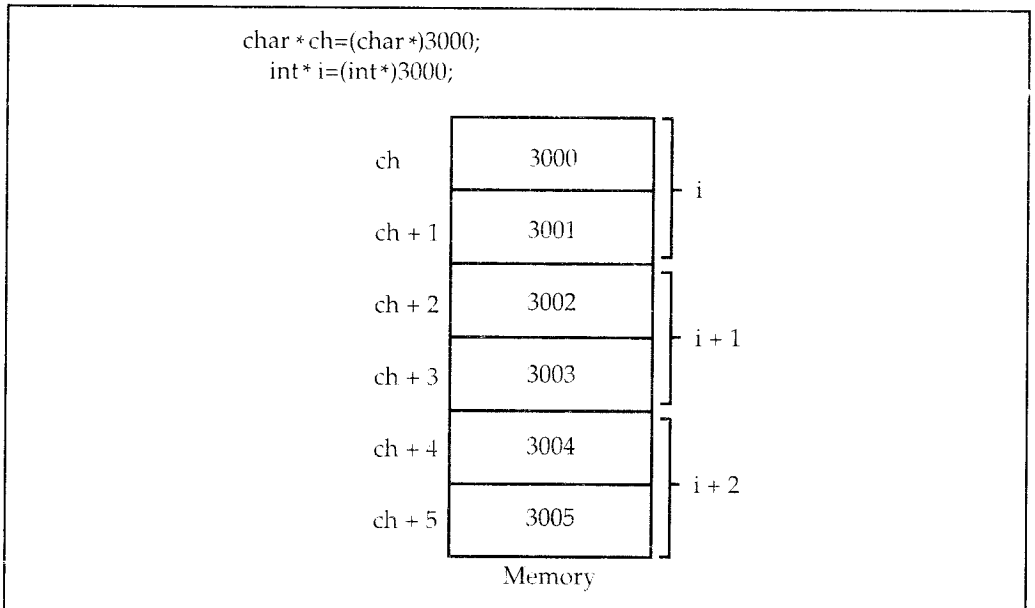


Figure 5-2. *All pointer arithmetic is relative to its base type (assume 2-byte integers)*

Pointer Comparisons

You can compare two pointers in a relational expression. For instance, given two pointers **p** and **q**, the following statement is perfectly valid:

```
if(p<q) printf("p points to lower memory than q\n");
```

Generally, pointer comparisons are used when two or more pointers point to a common object, such as an array. As an example, a pair of stack routines are developed that store and retrieve integer values. A stack is a list that uses first-in, last-out accessing. It is often compared to a stack of plates on a table—the first one set down is the last one to be used. Stacks are used frequently in compilers, interpreters, spreadsheets, and other system-related software. To create a stack, you need two functions: **push( )** and **pop( )**. The **push( )** function places values on the stack and **pop( )** takes them off. These routines are shown here with a simple **main( )** function to drive them. The program puts the values you enter into the stack. If you enter **0**, a value is popped from the stack. To stop the program, enter **–1**.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 50

void push(int i);
int pop(void);

int  *tos, *p1, stack[SIZE];

int main(void)
{
  int value;

  tos = stack; /* tos points to the top of stack */
  p1 = stack; /* initialize p1 */

  do {
    printf("Enter value: ");
    scanf("%d", &value);
    if(value!=0) push(value);
    else printf("value on top is %d\n", pop());
  } while(value!=-1);
```

```
  return 0;
}

void push(int i)
{
  p1++;
  if(p1==(tos+SIZE)) {
    printf("Stack Overflow.\n");
    exit(1);
  }
  *p1 = i;
}

int pop(void)
{
  if(p1==tos) {
    printf("Stack Underflow.\n");
    exit(1);
  }
  p1--;
  return *(p1+1);
}
```

You can see that memory for the stack is provided by the array **stack**. The pointer **p1** is set to point to the first element in **stack**. The **p1** variable accesses the stack. The variable **tos** holds the memory address of the top of the stack. It is used to prevent stack overflows and underflows. Once the stack has been initialized, **push( )** and **pop( )** may be used. Both the **push( )** and **pop( )** functions perform a relational test on the pointer **p1** to detect limit errors. In **push( )**, **p1** is tested against the end of stack by adding **SIZE** (the size of the stack) to **tos**. This prevents an overflow. In **pop( )**, **p1** is checked against **tos** to be sure that a stack underflow has not occurred.

In **pop( )**, the parentheses are necessary in the return statement. Without them, the statement would look like this:

```
return *p1 +1;
```

which would return the value at location **p1** plus one, not the value of the location **p1+1**.

Pointers and Arrays

There is a close relationship between pointers and arrays. Consider this program fragment:

```
char str[80], *p1;
p1 = str;
```

Here, **p1** has been set to the address of the first array element in **str**. To access the fifth element in **str**, you could write

```
str[4]
```

or

```
*(p1+4)
```

Both statements will return the fifth element. Remember, arrays start at 0. To access the fifth element, you must use 4 to index **str**. You also add 4 to the pointer **p1** to access the fifth element because **p1** currently points to the first element of **str**. (Recall that an array name without an index returns the starting address of the array, which is the address of the first element.)

The preceding example can be generalized. In essence, C/C++ provides two methods of accessing array elements: pointer arithmetic and array indexing. Although the standard array-indexing notation is sometimes easier to understand, pointer arithmetic can be faster. Since speed is often a consideration in programming, C/C++ programmers commonly use pointers to access array elements.

These two versions of **putstr( )**—one with array indexing and one with pointers—illustrate how you can use pointers in place of array indexing. The **putstr( )** function writes a string to the standard output device one character at a time.

```
/* Index s as an array. */
void putstr(char *s)
{
  register int t;

  for(t=0; s[t]; ++t) putchar(s[t]);
}
```

```
/* Access s as a pointer. */
void putstr(char *s)
{
  while(*s) putchar(*s++);
}
```

Most professional C/C++ programmers would find the second version easier to read and understand. In fact, the pointer version is the way routines of this sort are commonly written in C/C++.

Arrays of Pointers

Pointers may be arrayed like any other data type. The declaration for an **int** pointer array of size 10 is

```
int *x[10];
```

To assign the address of an integer variable called **var** to the third element of the pointer array, write

```
x[2] = &var;
```

To find the value of **var**, write

```
*x[2]
```

If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays—simply call the function with the array name without any indexes. For example, a function that can receive array **x** looks like this:

```
void display_array(int *q[])
{
  int t;

  for(t=0; t<10; t++)
    printf("%d ", *q[t]);
}
```

Remember, **q** is not a pointer to integers, but rather a pointer to an array of pointers to integers. Therefore you need to declare the parameter **q** as an array of integer pointers, as just shown. You cannot declare **q** simply as an integer pointer because that is not what it is.

Pointer arrays are often used to hold pointers to strings. You can create a function that outputs an error message given its code number, as shown here:

```
void syntax_error(int num)
{
  static char *err[] = {
    "Cannot Open File\n",
    "Read Error\n",
    "Write Error\n",
    "Media Failure\n"
  };

  printf("%s", err[num]);
}
```

The array **err** holds pointers to each string. As you can see, **printf( )** inside **syntax_error( )** is called with a character pointer that points to one of the various error messages indexed by the error number passed to the function. For example, if **num** is passed a 2, the message **Write Error** is displayed.

As a point of interest, note that the command line argument **argv** is an array of character pointers. (See Chapter 6.)

Multiple Indirection

You can have a pointer point to another pointer that points to the target value. This situation is called *multiple indirection*, or *pointers to pointers*. Pointers to pointers can be confusing. Figure 5-3 helps clarify the concept of multiple indirection. As you can see, the value of a normal pointer is the address of the object that contains the value desired. In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the object that contains the value desired.

Multiple indirection can be carried on to whatever extent rquired, but more than a pointer to a pointer is rarely needed. In fact, excessive indirection is difficult to follow and prone to conceptual errors.

Note *Do not confuse multiple indirection with high-level data structures, such as linked lists, that use pointers. These are two fundamentally different concepts.*

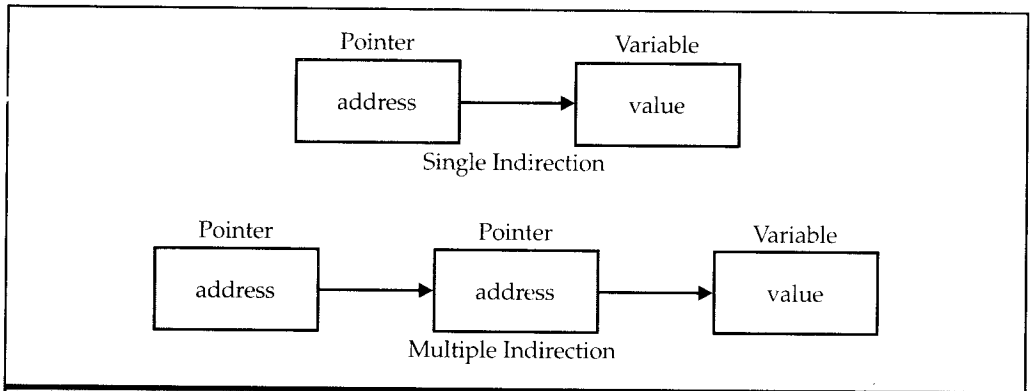


Figure 5-3. *Single and multiple indirection*

A variable that is a pointer to a pointer must be declared as such. You do this by placing an additional asterisk in front of the variable name. For example, the following declaration tells the compiler that **newbalance** is a pointer to a pointer of type **float**:

```
float **newbalance;
```

You should understand that **newbalance** is not a pointer to a floating-point number but rather a pointer to a **float** pointer.

To access the target value indirectly pointed to by a pointer to a pointer, you must apply the asterisk operator twice, as in this example:

```
#include <stdio.h>

int main(void)
{
  int x, *p, **q;

  x = 10;
  p = &x;
  q = &p;

  printf("%d", **q); /* print the value of x */

  return 0;
}
```

Here, **p** is declared as a pointer to an integer and **q** as a pointer to a pointer to an integer. The call to **printf( )** prints the number **10** on the screen.

Initializing Pointers

After a nonstatic local pointer is declared but before it has been assigned a value, it contains an unknown value. (Global and **static** local pointers are automatically initialized to null.) Should you try to use the pointer before giving it a valid value, you will probably crash your program—and possibly your computer's operating system as well—a very nasty type of error!

There is an important convention that most C/C++ programmers follow when working with pointers: A pointer that does not currently point to a valid memory location is given the value null (which is zero). By convention, any pointer that is null implies that it points to nothing and should not be used. However, just because a pointer has a null value does not make it "safe." The use of null is simply a convention that programmers follow. It is not a rule enforced by the C or C++ languages. For example, if you use a null pointer on the left side of an assignment statement, you still run the risk of crashing your program or operating system.

Because a null pointer is assumed to be unused, you can use the null pointer to make many of your pointer routines easier to code and more efficient. For example, you could use a null pointer to mark the end of a pointer array. A routine that accesses that array knows that it has reached the end when it encounters the null value. The **search( )** function shown here illustrates this type of approach.

```
/* look up a name */
int search(char *p[], char *name)
{
  register int t;

  for(t=0; p[t]; ++t)
    if(!strcmp(p[t], name)) return t;

    return -1; /* not found */
}
```

The **for** loop inside **search( )** runs until either a match is found or a null pointer is encountered. Assuming the end of the array is marked with a null, the condition controlling the loop fails when it is reached.

C/C++ programmers commonly initialize strings. You saw an example of this in the **syntax_error( )** function in the section "Arrays of Pointers." Another variation on the initialization theme is the following type of string declaration:

```
char *p = "hello world";
```

As you can see, the pointer **p** is not an array. The reason this sort of initialization works is because of the way the compiler operates. All C/C++ compilers create

what is called a *string table*, which is used to store the string constants used by the program. Therefore, the preceding declaration statement places the address of **hello world**, as stored in the string table, into the pointer **p**. Throughout a program, **p** can be used like any other string (except that it should not be altered). For example, the following program is perfectly valid:

```
#include <stdio.h>
#include <string.h>

char *p = "hello world";

int main(void)
{
  register int t;

  /* print the string forward and backwards */
  printf(p);
  for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);

  return 0;
}
```

In Standard C++, the type of a string literal is technically **const char ***. But C++ provides an automatic conversion to **char ***. Thus, the preceding program is still valid. However, this automatic conversion is a deprecated feature, which means that you should not rely upon it for new code. For new programs, you should assume that string literals are indeed constants and the declaration of **p** in the preceding program should be written like this.

```
const char *p = "hello world";
```

Pointers to Functions

A particularly confusing yet powerful feature of C++ is the *function pointer*. Even though a function is not a variable, it still has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

You obtain the address of a function by using the function's name without any parentheses or arguments. (This is similar to the way an array's address is obtained

when only the array name, without indexes, is used.) To see how this is done, study the following program, paying close attention to the declarations:

```
#include <stdio.h>
#include <string.h>

void check(char *a, char *b,
           int (*cmp)(const char *, const char *));

int main(void)
{
  char s1[80], s2[80];
  int (*p)(const char *, const char *);

  p = strcmp;

  gets(s1);
  gets(s2);

  check(s1, s2, p);

  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
```

When the **check( )** function is called, two character pointers and one function pointer are passed as parameters. Inside the function **check( )**, the arguments are declared as character pointers and a function pointer. Notice how the function pointer is declared. You must use a similar form when declaring other function pointers, although the return type and parameters of the function may differ. The parentheses around the ***cmp** are necessary for the compiler to interpret this statement correctly.

Inside **check( )**, the expression

```
(*cmp)(a, b)
```

calls **strcmp( )**, which is pointed to by **cmp**, with the arguments **a** and **b**. The parentheses around ***cmp** are necessary. This is one way to call a function through a pointer. A second, simpler syntax, as shown here, may also be used.

```
cmp(a, b);
```

The reason that you will frequently see the first style is that it tips off anyone reading your code that a function is being called through a pointer. (That is, that **cmp** is a function pointer, not the name of a function.) Other than that, the two expressions are equivalent.

Note that you can call **check( )** by using **strcmp( )** directly, as shown here:

```
check(s1, s2, strcmp);
```

This eliminates the need for an additional pointer variable.

You may wonder why anyone would write a program in this way. Obviously, nothing is gained and significant confusion is introduced in the previous example. However, at times it is advantageous to pass functions as parameters or to create an array of functions. For example, when a compiler or interpreter is written, the parser (the part that evaluates expressions) often calls various support functions, such as those that compute mathematical operations (sine, cosine, tangent, etc.), perform I/O, or access system resources. Instead of having a large **switch** statement with all of these functions listed in it, an array of function pointers can be created. In this approach, the proper function is selected by its index. You can get the flavor of this type of usage by studying the expanded version of the previous example. In this program, **check( )** can be made to check for either alphabetical equality or numeric equality by simply calling it with a different comparison function.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

void check(char *a, char *b,
           int (*cmp)(const char *, const char *));
int numcmp(const char *a, const char *b);

int main(void)
{
  char s1[80], s2[80];
```

```
  gets(s1);
  gets(s2);

  if(isalpha(*s1))
         check(s1, s2, strcmp);
  else
         check(s1, s2, numcmp);

  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}

int numcmp(const char *a, const char *b)
{
  if(atoi(a)==atoi(b)) return 0;
  else return 1;
}
```

In this program, if you enter a letter, **strcmp( )** is passed to **check( )**. Otherwise, **numcmp( )** is used. Since **check( )** calls the function that it is passed, it can use different comparison functions in different cases.

C's Dynamic Allocation Functions

Pointers provide necessary support for C/C++'s dynamic allocation system. *Dynamic allocation* is the means by which a program can obtain memory while it is running. As you know, global variables are allocated storage at compile time. Local variables use the stack. However, neither global nor local variables can be added during program execution. Yet there will be times when the storage needs of a program cannot be known ahead of time. For example, a program might use a dynamic data structure, such as a linked list or binary tree. Such structures are inherently dynamic in nature, growing or shrinking as needed. To implement such a data structure requires that a program be able to allocate and free memory.

C++ actually supports two complete dynamic allocation systems: the one defined by C and the one specific to C++. The system specific to C++ contains several improvements over that used by C, and this approach is discussed in Part Two. Here, C's dynamic allocation functions are described.

Memory allocated by C's dynamic allocation functions is obtained from the *heap*—the region of free memory that lies between your program and its permanent storage area and the stack. Although the size of the heap is unknown, it generally contains a fairly large amount of free memory.

The core of C's allocation system consists of the functions **malloc( )** and **free( )**. (Most compilers supply several other dynamic allocation functions, but these two are the most important.) These functions work together using the free memory region to establish and maintain a list of available storage. The **malloc( )** function allocates memory and the **free( )** function releases it. That is, each time a **malloc( )** memory request is made, a portion of the remaining free memory is allocated. Each time a **free( )** memory release call is made, memory is returned to the system. Any program that uses these functions should include the header file **stdlib.h**. (A C++ program may also use the C++-style header **<cstdlib>**.)

The **malloc( )** function has this prototype:

```
void *malloc(size_t number_of_bytes);
```

Here, *number_of_bytes* is the number of bytes of memory you wish to allocate. (The type **size_t** is defined in **stdlib.h** as, more or less, an **unsigned** integer.) The **malloc( )** function returns a pointer of type **void ***, which means that you can assign it to any type of pointer. After a successful call, **malloc( )** returns a pointer to the first byte of the region of memory allocated from the heap. If there is not enough available memory to satisfy the **malloc( )** request, an allocation failure occurs and **malloc( )** returns a null.

The code fragment shown here allocates 1,000 bytes of contiguous memory:

```
char *p;
p = malloc(1000); /* get 1000 bytes */
```

After the assignment, **p** points to the start of 1,000 bytes of free memory.

In the preceding example, notice that no type cast is used to assign the return value of **malloc( )** to **p.** In C, a **void *** pointer is automatically converted to the type of the pointer on the left side of an assignment. However, it is important to understand that this automatic conversion *does not* occur in C++. In C++, an explicit type cast is needed when a **void *** pointer is assigned to another type of pointer. Thus, in C++, the preceding assignment must be written like this:

```
p = (char *) malloc(1000);
```

As a general rule, in C++ you must use a type cast when assigning (or otherwise converting) one type of pointer to another. This is one of the few fundamental differences between C and C++.

The next example allocates space for 50 integers. Notice the use of **sizeof** to ensure portability.

```
int *p;
p = (int *) malloc(50*sizeof(int));
```

Since the heap is not infinite, whenever you allocate memory, you must check the value returned by **malloc( )** to make sure that it is not null before using the pointer. Using a null pointer will almost certainly crash your program. The proper way to allocate memory and test for a valid pointer is illustrated in this code fragment:

```
p = (int *) malloc(100);
if(!p) {
  printf("Out of memory.\n");
  exit(1);
}
```

Of course, you can substitute some other sort of error handler in place of the call to **exit( )**. Just make sure that you do not use the pointer **p** if it is null.

The **free( )** function is the opposite of **malloc( )** in that it returns previously allocated memory to the system. Once the memory has been freed, it may be reused by a subsequent call to **malloc( )**. The function **free( )** has this prototype:

```
void free(void *p);
```

Here, *p* is a pointer to memory that was previously allocated using **malloc( )**. It is critical that you *never* call **free( )** with an invalid argument; otherwise, you will destroy the free list.

Problems with Pointers

Nothing will get you into more trouble than a wild pointer! Pointers are a mixed blessing. They give you tremendous power and are necessary for many programs. At the same time, when a pointer accidentally contains a wrong value, it can be the most difficult bug to find.

An erroneous pointer is difficult to find because the pointer itself is not the problem. The problem is that each time you perform an operation using the bad pointer, you are reading or writing to some unknown piece of memory. If you read from it, the worst that can happen is that you get garbage. However, if you write to it, you might be writing over other pieces of your code or data. This may not show up until later in the execution of your program, and may lead you to look for the bug in the wrong place. There may be little or no evidence to suggest that the pointer is the original cause of the problem. This type of bug causes programmers to lose sleep time and time again.

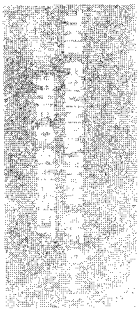
Because pointer errors are such nightmares, you should do your best never to generate one. To help you avoid them, a few of the more common errors are discussed here. The classic example of a pointer error is the *uninitialized pointer*. Consider this program.

```
/* This program is wrong. */
int main(void)
{
  int x, *p;

  x = 10;
  *p = x;

  return 0;
}
```

This program assigns the value 10 to some unknown memory location. Here is why: Since the pointer **p** has never been given a value, it contains an unknown value when the assignment ***p = x** takes place. This causes the value of **x** to be written to some unknown memory location. This type of problem often goes unnoticed when your program is small because the odds are in favor of **p** containing a "safe" address—one that is not in your code, data area, or operating system. However, as your program grows, the probability increases of **p** pointing to something vital. Eventually, your program stops working. The solution is to always make sure that a pointer is pointing at something valid before it is used.

A second common error is caused by a simple misunderstanding of how to use a pointer. Consider the following:

```
/* This program is wrong. */
#include <stdio.h>

int main(void)
{
```



```
    int x, *p;

    x = 10;
    p = x;

    printf("%d", *p);

    return 0;
}
```

The call to **printf( )** does not print the value of **x**, which is 10, on the screen. It prints some unknown value because the assignment

```
p = x;
```

is wrong. That statement assigns the value 10 to the pointer **p**. However, **p** is supposed to contain an address, not a value. To correct the program, write

```
p = &x;
```

Another error that sometimes occurs is caused by incorrect assumptions about the placement of variables in memory. You can never know where your data will be placed in memory, or if it will be placed there the same way again, or whether each compiler will treat it in the same way. For these reasons, making any comparisons between pointers that do not point to a common object may yield unexpected results. For example,

```
char s[80], y[80];
char *p1, *p2;

p1 = s;
p2 = y;
if(p1 < p2) . . .
```

is generally an invalid concept. (In very unusual situations, you might use something like this to determine the relative position of the variables. But this would be rare.)

A related error results when you assume that two adjacent arrays may be indexed as one by simply incrementing a pointer across the array boundaries. For example,

```
int first[10], second[10];
int *p, t;
```

```
p = first;
for(t=0; t<20; ++t)  *p++ = t;
```

This is not a good way to initialize the arrays **first** and **second** with the numbers 0 through 19. Even though it may work on some compilers under certain circumstances, it assumes that both arrays will be placed back to back in memory with **first** first. This may not always be the case.

The next program illustrates a very dangerous type of bug. See if you can find it.

```
/* This program has a bug. */
#include <string.h>
#include <stdio.h>

int main(void)
{
  char *p1;
  char s[80];

  p1 = s;
  do {
    gets(s);  /* read a string */

    /* print the decimal equivalent of each
       character */
    while(*p1) printf(" %d", *p1++);

  } while(strcmp(s, "done"));

  return 0;
}
```

This program uses **p1** to print the ASCII values associated with the characters contained in **s**. The problem is that **p1** is assigned the address of **s** only once. The first time through the loop, **p1** points to the first character in **s**. However, the second time through, it continues where it left off because it is not reset to the start of **s**. This next character may be part of the second string, another variable, or a piece of the program! The proper way to write this program is

```
/* This program is now correct. */
#include <string.h>
```

```
#include <stdio.h>

int main(void)
{
  char *p1;
  char s[80];

  do {
    p1 = s;
    gets(s);  /* read a string */

    /* print the decimal equivalent of each
       character */
    while(*p1) printf(" %d", *p1++);

  } while(strcmp(s, "done"));

  return 0;
}
```

Here, each time the loop iterates, **p1** is set to the start of the string. In general, you should remember to reinitialize a pointer if it is to be reused.

The fact that handling pointers incorrectly can cause tricky bugs is no reason to avoid using them. Just be careful, and make sure that you know where each pointer is pointing before you use it.