Pointers

A *pointer* is an address of a byte in main memory. Pointers are widely used in C++ to facilitate efficient dynamic processing of data.

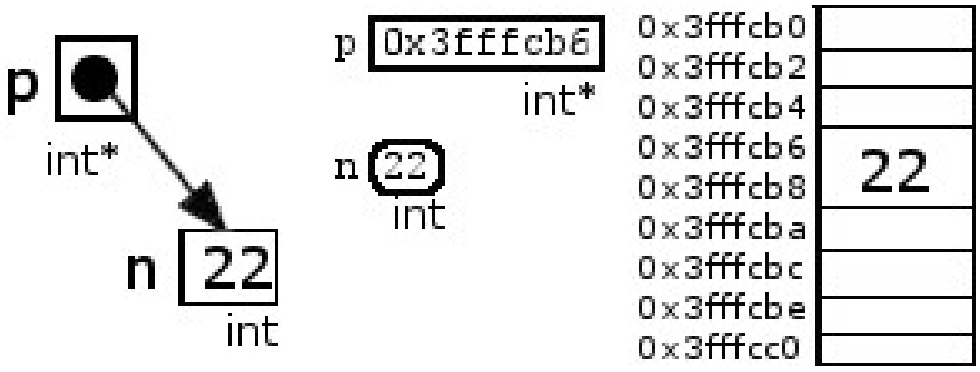
1 POINTERS

If `T` is any type, then `T*` is the derived type"pointer to `T`." The base type `T` may be a built-in type, such as `int` or `float`. Or it may be a user-defined type, such as `Date` or `Person`.

EXAMPLE 1. A Pointer to an `int`

```
int n => 22; // defines the integer n initialized to 22
int* p = &n; // defines the pointer p initialized to the address of n
```

This code defines two objects: the integer `n` and the pointer `p`. Both are initialized: `n` with the value 22, and p with the value `&n` which means the memory address of the object `n`. We think of `p` as an arrow that points to the object n, as shown here on the left. But really `p` is an object whose value is a memory address, perhaps the hexadecimal address `0x3fffcfb6`, as shown above on the right. Note that the pointer contains only the address of the first byte occupied by n. In this example, `n` is an integer occupying the four bytes `0x3fffcb6` to `0x3fffcba`, but `p` contains only the address of the first byte `0x3fffcb6`.



2 THE DEREFERENCE OPERATOR

The asterisk symbol `*` has two related uses in C++. When used as a suffix on a type, as in Example 1, it defines the pointer type derived from the base type. It is also used as a prefix to the name of a pointer. In that context it is called the *dereference operator*. The resulting expression refers to the object to which the pointer points. This is called *dereferencing* the pointer.

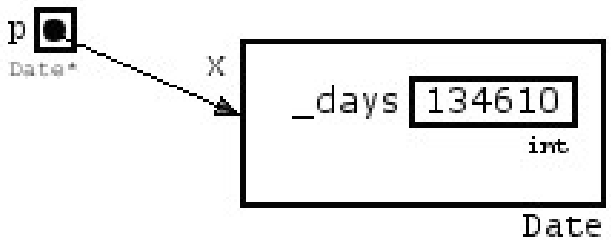
EXAMPLE 8.2 Dereferencing a Pointer

```
Date x(1969,7,20); // the object x represents the date Jul 20, 1969
Date* p = &x; / the pointer p points to the object x
cout << "Man stepped onto the moon on " << *p << endl;
```

Here `p` points to a `Date` object, where `Date` is a user-defined class. The pointer is dereferenced in the output statement: `*p` is the same as `x`, so `x` is inserted into the output stream, thereby printing

```
Man stepped onto the moon on 1969-07-20
```

The relationship between p and x is illustrated in the picture at right. Note that p has type Date* and x has type Date.



When a pointer points to a class object, as in Example 2 above, there is an alternative notation for calling a member function bound to a dereferenced pointer. The following two forms are equivalent:

```
(*P).f();
p->f();
```

The second of these two equivalent forms is generally preferred because it is simpler and because the combination symbol -> suggests the pointer relationships.

EXAMPLE 8.3 Binding a Function Call to a Dereferenced Pointer

This code continues from Example 8 2:

```
Date y(1972,12,17);
Date* q = &y;
cout << "Apollo 17 left the moon on " << q->weekday() << ","
     << q->month() << "/" << (*q).day() << "/" << y.year() << endl;
cout << "The era of moon landings lasted " << y - *p << " days.";
```

The output is

```
Apollo 17 left the moon on Sun, 12/17/1972 The era of moon landings lasted 1246 days.
```

3 POINTER OPERATIONS

Pointer values may be output with the insertion operator <<.

EXAMPLE 8.4 Pointer I/O

```
Date x (1941,12,7); // the object x represents the date Dec 7 1941
Date* p = &x; // the pointer p points to the object x
cout << x << ‘\t’ << p << ‘\t’ << *p << endl;
```

The output is

```
1941-12-07 0x3fffcc4 1941-12-07
```

This shows that the Date object x is stored in memory beginning at byte number 0x3fffcc4.

However, pointers cannot be input:

```
cin >> p; // ERROR; the insertion operator is not defined for pointers
```

Pointers may be assigned to other pointers of the same type.

EXAMPLE 8.5 Assigning Pointers

This code continues from Example 8.3:

```
q = p; // now both p and q point to x
cout << "Apollo 11 landed on the moon on " << q->weekday() << ", "
      << q->month() << ‘‘/" << q->day() << ‘‘/" << q->year() << endl;
```

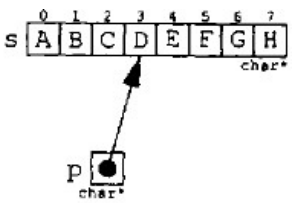
The output is

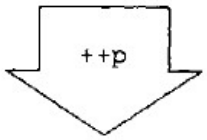
```
Apollo 11 landed on the moon on Sun, 7/20/1969
```

Pointers can also be incremented and decremented.

EXAMPLE 8.6 Pointer Arithmetic

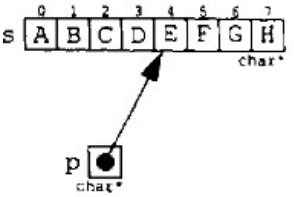
```
char S[] = "ABCDEFGH";
char* p = &S[3]; //p points to s[3]
cout << "*p = " << *p << endl;
++p; // p points to s[4]
cout << "*p = " << *p << endl;
p += 3; // p points to s[7]
cout << "*p = " << *p << endl;
p -= 6; //p points to s[1]
cout << "*p = " << *p << endl;
```



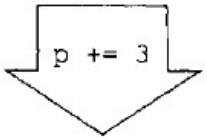


Here is the output:

```
*p = D
*p = E
*p = H
*p = B
```



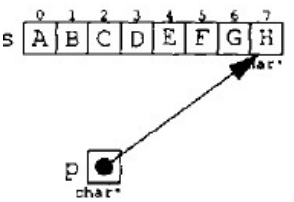
The pointer is initialized to point to `s[3]` which contains `'D'`. Incrementing `p` advances it to point to `s[4]` which contains `'E'`. Adding 3 to `p` advances it to point to `s[7]`, which contains `'H'`. Subtracting 6 from `p` moves it back to point to `s[1]` which contains `'B'`.

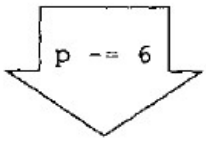


Pointers can also be subtracted from other pointers:

EXAMPLE 8.7 Subtracting Pointers

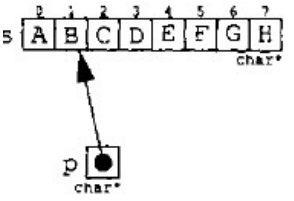
```
char s[] = "ABCDEFGHIJ";
char* p = &s[3];
char* q = &s[6];
cout << "*p = " << *p << "*q = " << *q;
--p; // p points to s[2]
++q; // q points to s[7]
cout << "\t*p = " << *p << ", " << "*q = " << *q;
cout << "\tp - q = " << q - p << endl;
```





The output is:

```
*p = D, *q = G
*p = C, *q = H
q - p = 5
```



The pointer `p` is initialized to point to `s[3]` which contains `'D'`, and then the pointer `q` is initialized to point to `s[6]` which contains `'G'`. Decrementing `p` makes it point to `s[2]` which contains `'C'`, and then incrementing `q` makes it point to `s[7]` which contains `'H'`. Now `q` contains an address which is 5 bytes higher than the address in `p`, so `q - p` evaluates to 5.

Arithmetic on pointers depends upon the size of their base types. Incrementing a pointer to `char` increases its value by 1. But incrementing a pointer to `double` increases its value by 8. In general the unit used in arithmetic on pointers of type pointer to `T` is `sizeof(T)`.

EXAMPLE 8.8 Pointer Arithmetic with Unit Size 4

```
int a[] = {22, 33, 44, 55, 66, 77, 88, 99 } ;
int* p = &a[3]; // p points to a[3]
cout << "p = " << p << ", *p = " << *p << endl;
++p; // p points to a [4]
cout << "p = " << p ", *p = " << *p << endl;
p += 3; // p points to a[7]
cout << "p = " << p << ", *p = " << *p << endl;
p -= 6; // p points to a [1]
cout << "p = " << p << ", *p = " << *p << endl;
```

The output is

```
p = 0x3fffcb4, *p = 55
p = 0x3fffcb8, *p = 66
p = 0x3fffcc4, *p = 99
p = 0x3fffcac, *p = 33
```

The pointer `p` is initialized to point to `a[3]` which contains 55. Incrementing `p` advances it to point to `a[4]` which contains 66. This changes the value of `p` from `0x3fffcb4` to `0x3fffcb8`, an increase of 4. Adding 3 to `p` advances it to point to `a[7]` which contains 99. This changes the value of `p` from `0x3fffcb8` to `0x3fffcc4`, an increase of 12. Subtracting 6 from `p` moves it back to point to `a[1]` which contains 33. This changes the value of `p` from `0x3fffcc4` to `0x3fffcac`, a decrease of 24.

Note that pointers in the same expression must point to the same type:

```
int n = 22;
double x = 3.141592653589793;
int* p = &n;
double* q = &x;
cout << p << endl; // ok
cout << q << endl; // ok
cout << q - p << endl; // ILLEGAL: *p and *q have different types
```

Here `p` has type pointer to `int` and `q` has type pointer to `double`.

A pointer expression such as `p+5` makes sense and acts like a pointer. In particular, it can be dereferenced: `*(p+5)` refers to the object located at address `p + 5*sizeof(T)`, where `T` is the base type for the pointer `p`. If `p` has type pointer to `short`, then `p+5` points to the address `p+10`; but if `p` has type pointer to `double`, then `p+5` points to the address `p+40`.

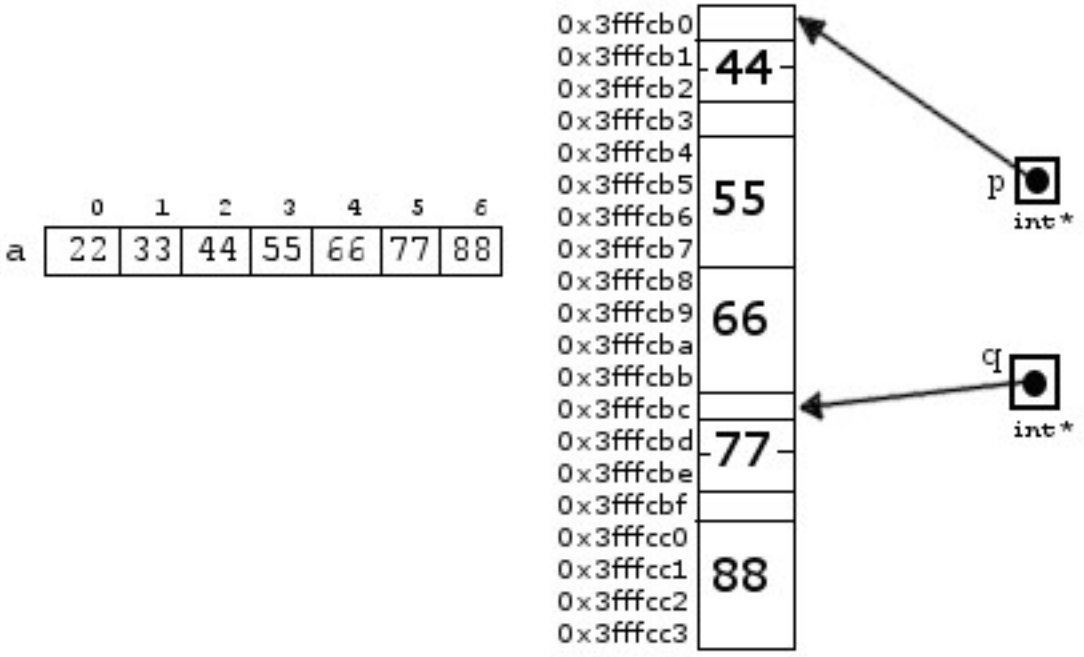
EXAMPLE 8.9

```
int a[] = { 22, 33, 44, 55, 66, 77, 88, 99} ;
int* p = &a[2]; // points to a[2]
cout << "p = " << p << ", *p = " << *p << endl;
int* q = p+3; // q points to a[5]
cout << "q = " << q << ", *q = " << *q << endl;
cout << "p+4 = " << p+4 << ", *(p+4) = " << *(p+4) << endl;
cout << "q-2 = " << q-2 << ", *(q-2) = " << *(q-2) << endl;
cout << "q - p = " << q - p << endl;
```

The output is

```
p = 0x3fffcb0, *p = 44
q = 0x3fffcbc, *q = 77
p+4 = 0x3fffcc0, *(p+4) = 88
q-2 = 0x3fffcb4,*(q-2) = 55
q - p = 3
```

The pointer `p` is initialized to point to `a[2]` which contains 44 at address `0x3fffcb0`. Then the pointer `q` is initialized to point to `a[5]` which contains 77 at address `0x3fffcbc`. Note that this is 12 bytes farther than `0x3fffcb0`. The expression `p+4` evaluates to `0x3fffcc0` which is 16 bytes farther than `0x3fffcb0`. It is the first of the four bytes that contain `a[6]` which is 88, so the dereferenced expression `*(p+4)` evaluates to 88. Similarly, the expression `q-2` evaluates to `0x3fffcb4` which is 8 bytes ahead of `0x3fffcbc`; that is the first of the four bytes that contain `a[3]` which is 55, so the dereferenced expression `*(q-2)` evaluates to 55. Finally, the expression `q - p` evaluates to 3 because the distance from `0x3fffcb0` to `0x3fffcbc` is 3 four-byte units:



The array a is pictured above on the left. The diagram on the right shows a detail of memory where the array is stored. The pointer `p` points to byte number `0x3fffcb0` which is the first of the four bytes that hold `a[2]`, and the pointer `q` points to byte number `0x3fffcbc` which is the first of the four bytes that hold `a[5]`.

THE REFERENCE OPERATOR

Like the asterisk symbol `*`, the ampersand symbol `&` has two related uses in C++. When used as a prefix to the name of an object, it is called the reference operator. As the previous examples illustrate, the reference operator returns the address of the object:

```
p = &y; // assigns the address of y to the pointer p
```

But we have also used the ampersand as suffix to a type:

```
void swap(float& x, float& y);
```

When used this way, the ampersand defines a derived type, called a reference type. For example, floats is the type "reference to float." This is the way reference parameters are declared in functions.
A reference is a synonym or alias; i.e., another name for an existing object. Chapter 4 describes how references

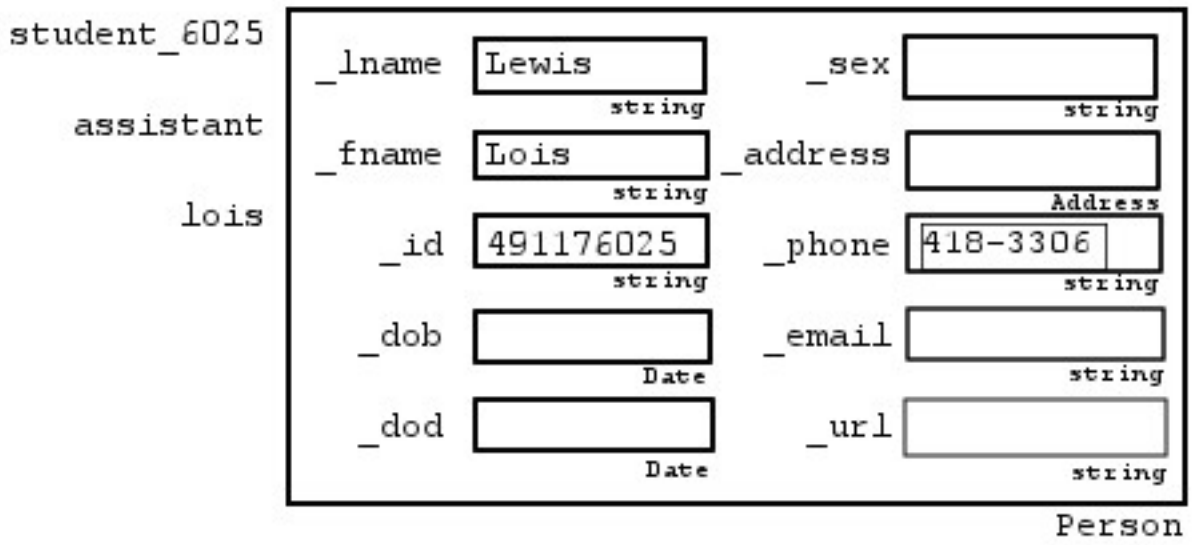
are used for passing to and returning from functions. References are also used independently of functions.

EXAMPLE 8.10 Declaring References

This code uses the Person class defined on page in chapter 7:

```
Person student_6025(‘‘Lewis", ‘‘Lois", ‘‘491176025");
Person& assistant = student_6025;
Person& lois = assistant;
lois.set_phone(‘‘418-3306");
```

It defines a single Person object with three different names: `student_6025`, `assistant`, and `lois`. Then it sets the phone number for that single object:



Like constants, references must be initialized. That should seem reasonable, since you couldn't have an alternative name for something that doesn't exist. Although not required, pointers should also be initialized.
The reference and dereference operators are inverses in the sense that each reverses the action of the other.

EXAMPLE 8.11 The Reference and Dereference Operators

```
Address x(‘‘72 N Main", ‘‘Troy", ‘‘NY", ‘‘12180", ‘‘USA");
Address y(‘‘49 Elm St", ‘‘Troy", ‘‘MI", ‘‘48099", ‘‘USA");
Address* p = &x; // p points to x
Address* q = &y; // q points to y
Address z = *p; // initializes z to x
z = *&y; // assigns y to z
p = &*q; // assigns q to p
```

Note that although the reference operator can be applied to any lvalue, the dereference operator can be applied only to pointers:

```
Address** pp = &p; // ok:  p has type Address*
z = *x; // ERROR: x is not a pointer
```

8.5 NULL POINTERS

The number zero (0) is an integer literal that can be used as a pointer value. But as an address, 0x0 would locate the first byte in memory, which is certainly outside any memory segment allocated to a program. So 0 is a valid pointer value that cannot be dereferenced. It is called the null value, and any pointer whose value is zero is called a null pointer.

Null pointers are often used to indicate the end of something, like a list or a tree. But since they cannot be dereferenced, they are a common cause of run-time error.

EXAMPLE 8.12 Bus Errors

This program defines a version of the Standard C `strchr()` function for finding characters within C-strings. The `locate()` function prints the index of the given character within the given string if it is found.

But if the character is not in the string, the program crashes on the dereference `*p` because in that case `p` is the null pointer:

```
1  char* strchr(char* s, char c) {
       for(char* p=s; *p; p++)
3         if (*p == c) return p;
       return 0;
5  }

7  void locate(char* s, char c) {
       char* p = strchr(s, c);
9      if (*p == 0) cout << "Not found"
                         << endl; // ERROR
11     else cout << "The first occurrence of the character '"
                 << c << "'\n\tin the string << s
13               << "\"\n\tis at position: " << p - s << endl;
   }
15
   int main() {
17     locate("Newton, Isaac, 1642-1727", 'a');
       locate ("Leibniz, Gottfried Wilhelm, 1646-1716", '1');
19     locate("Gauss, Carl Friedrich, 1777-1855", 'k');
   }
```

The output is

```
The first occurrence of the character 'a'
        in the string "Newton, Isaac, 1642-1727"
        is at position: 10
The first occurrence of the character '1'
        in the string "Leibniz, Gottfried Wilhelm, 1646-1716"
        is at position: 21
Bus error
```

On the third call to the `locate()` function, the `strchr()` function returns 0 to p. Since the dereference `*p` is not possible when `p` is 0, the program crashes with a "Bus error." This code is repaired simply by changing the condition to `(p == 0)`.

Notice how the for loop works in the `strchr()` function. It is controlled by the pointer `p` which traverses the string `s`. The C-string variable `s` contains the address of the first character in the string, so the loop is initialized by setting `p = s`. The characters in the string reside in consecutive bytes in memory, so `++p` moves it down the

string one character at a time. And since every C-string ends with the null character 0, the condition `*p != 0` can be used to continue the loop. But the condition `*p != 0` is equivalent to the condition `*p` because any non-zero integer value (or pointer values) is always interpreted as the `bool` value `true`, and the zero integer value (or null pointer) is always interpreted as the `bool` value `false`. Thus, the form

```
for (char* p=s; *p; p++)
```

neatly and succinctly traverses the C-string, giving access to each character of `s` through `*p`. This is a standard technique in C programs. The bus error illustrated in Example 12 was caused by dereferencing the null pointer. A similar run-time error occurs when a dangling pointer is dereferenced. In that case, the diagnostic error message from the operating system is likely to be "Segmentation fault" because the dereference is an attempt to access a memory location that is outside the segment allocated to the running process.

8.6 DYNAMIC ARRAYS

A dynamic array is an array whose size can be changed dynamically while the program is running. We used dynamic arrays to implement the `Stack`, `Queue`, and `Deque` classes in Chapter 7.

EXAMPLE 8.13 Using a Dynamic Array in the `Stack` Class

Here are the relevant parts of the `Stack` class defined in Chapter 7:

```
class Stack {
public:
  Stack(int s=100); // sets the default maximum number at 100
  ~Stack();
  //...
private:
   char* _a;  // the stack itself: a dynamic array of char
   int _max;  // the maximum number of elements on the stack
   int _count; // the number of elements on the stack
};

Stack::Stack(int m) : _max(m), _count(0) {
   _a = new char[_max];
   assert(_a != 0);
}

Stack::~Stack() {
   delete [] a;
}
```

The dynamic array `_a` is declared to be a `char*` because the array's element type is `char`. If the class were for stacks of `Person` objects instead of stacks of `char`s, we would declare its dynamic array as

```
Person* _a;
```

The default constructor uses the `new` operator to allocate space for m elements of type `char` to the array `_a`. This storage allocation occurs dynamically, at the moment a declaration such as

```
Stack stack(500); // creates a stack that can hold up to 500 chars
```

executes. Those 500 bytes of memory remain allocated to `_a` until the stack goes out of scope. At that moment, the destructor is invoked, which uses the delete operator to deallocate the storage. Notice the call

```
assert(_a != 0);
```

in the constructor. This is done to check whether the storage allocation was successful. If the operating system is unable to allocate the number of bytes requested, it sets the pointer `_a` to 0. That would likely happen upon a declaration such as

```
Stack big_stack(5000000); // asks for 5 MB of memory
```

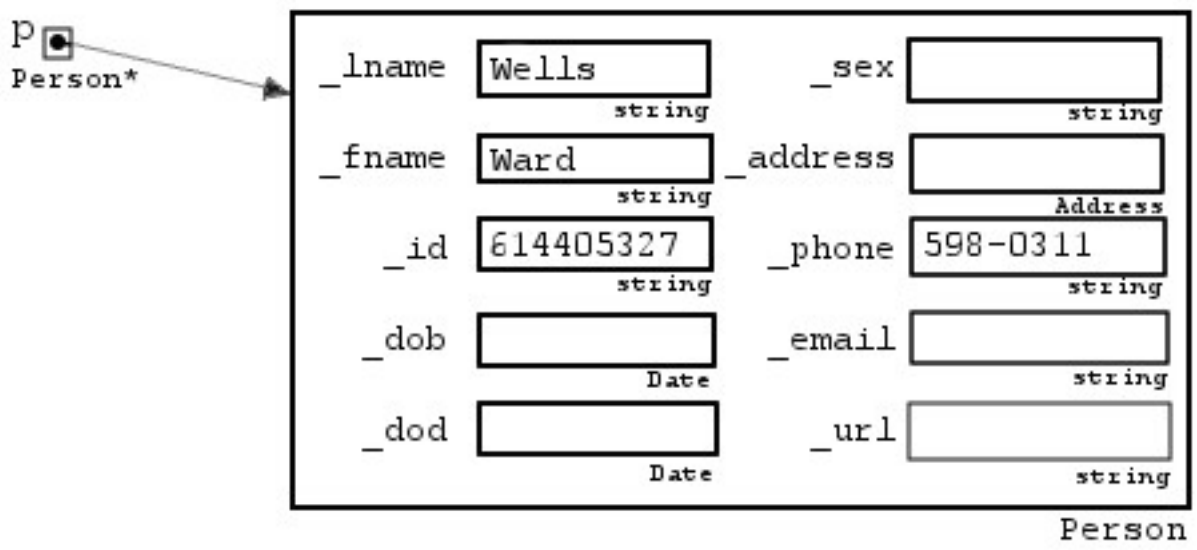
The `new` and `delete` operators are used to allocate and deallocate dynamic arrays. But they can also be used to

allocate and deallocate individual objects.

EXAMPLE 8.14 Dynamic Objects

This example uses the `Person` and `Date` classes defined earlier.

```
Person* p = new Person("Wells", "Ward", "614405927");
p->set_dob(Date(1980,8,18));
```



The first line uses the `new` operator to create an anonymous `Person` object. It initializes its `_lname`, `_fname`, and `_id` fields, and defines the pointer `p` to point to it. The second line dereferences the pointer `p` to call the `set_dob` member function to set the object's `_dob` field.

Note how the `Date` constructor is used in the second line to create an anonymous temporary `Date` object to represent the date Aug 18 1980. That object is passed to `set_dob()` member function of the `Person` class to set the `Person` object's `_dob` field. The `Date` object is passed by value, so the `Date` class copy constructor copies the data into the `_dob` field.

8.7 THE `this` POINTER

When a class member function is called, it must be bound to an instance of the class. For example,

```
Person sara(‘‘Smith", ‘‘Sara", ‘‘510880457");
sara.set_email(‘‘ssmith@richmond.edu");
```

The call `set_email (‘‘ssmitharichmond.edu")` is bound to the object `sara`. To carry out its instructions, the function needs access to both the explicit argument "ssmith@richmond.edu" and the implicit argument `sara`. Member functions can access their explicit arguments by means of their parameters. Member functions can access their implicit argument by means of the `this` pointer.

The C++ keyword `this` can be used only within member functions. It is a predefined pointer that always points to the object to which the member function call is bound.

EXAMPLE 8.15 Using the `this` Pointer

This member function for the `Purse` class defined in Chapter 7 determines whether a variable is a synonym for its implicit argument

```
1 bool is_same(Purse& x) {
```

```
    return bool(&x == this);
}
```

If x and y are Purse objects, this function could be called like this:

```
if(x.is_same(y))
    cout >>"It's the same purse.\n";
```

The function returns true if the address of y is the same as this which points to x; *i.e.*, if x and y have the same address. That condition determines, by definition, whether x and y are names for the same object. Of course, this function is unnecessary. The call could be replaced by the condition (&x == &y).

Whenever a class member function has to return a reference to its implicit argument, it should return *this; That is the standard code for the overloaded assignment operator.

EXAMPLE 8.16 An Assignment Operator for the Stack Class

This member function could be added to the Stack class defined earlier to allow the assignment of one stack to another:

```
Stack& Stack::operator=(const Stack& x) {
    if (&x == this) return *this;
    _max = x._max;
    _count = x._count;
    _a = new char[_max];
    for (int i = 0; i < _count; i++ )
     a[i] = x._a[i];
    return *this;
}
```

The function returns immediately if its explicit argument is the same object as its implicit argument. Otherwise, it copies the values of _max and _count, allocates the dynamic array _a, and then copies the stack elements from x to *this.

The assignment operator returns *this so that the operator can be used in a cascade assignment, like this:

```
z = y = x;
```

This calls the assignment operator twice, like this:

```
z.operator=(y.operator=(x));
```

The inside call assigns x to y and returns a reference to y. Then the outside call uses that reference as its explicit argument, and assigns y to z.

Warning: Without the explicit definition of the class assignment operator, the compiler will generate a default version of it. But this default version simply performs a bitwise copy of an object's data members. For classes like Person and Purse, a bitwise copy is completely adequate, so there is no need to include an explicit definition of the assignment operator. However, for classes like Stack whose member data are dynamic, a bitwise copy produces incorrect results. A bitwise copy of the data member a will not duplicate the array; it simply duplicates its name. So the assignment of one Stack object to another would result in two separate objects using the same array to

hold their data.

Any class whose member data use pointers should either include explicit definitions of the class's copy constructor and assignment operator, or those two member functions should be disabled by declaring them to be `private`, like this:

```
class Stack {
public:
   Stack(int s=100); // sets the default maximum number at 100
   ~Stack();
   //...
private:
   char* _a;  // the stack itself: a dynamic array of char
   int _max;  // the maximum number of elements on the stack
   int _count; // the number of elements on the stack
   Stack(const Stack& x) { }
   Stack& operator=(const Stack& x) { return *this; }
};
```

As `private` function members, they can be called only from within the class itself.

1. What is the difference between `p` and `*p`? (Assume that `p` is a pointer type.)
2. What is the difference between `x` and `&x`?
3. What is the difference between `++(*p)` and `*(++p)`?
4. Why must a reference be initialized?
5. What boolean expression determines whether two names, x and y, are names for the same object?
6. What is a dangling pointer?
7. What's wrong with storing pointers in a file?
8. What is the difference between a static array and a dynamic array?
9. How are dynamic arrays better than static arrays?
10. What is the `this` pointer?
11. Why should the assignment operator of a class return `*this`?
12. Why is it important to define the copy constructor and the assignment operator explicitly `private` in a class whose data members include pointers, instead of allowing the compiler to generate its default versions of these two member functions?
13. What does the `new` operator do?
14. What does the `delete` operator do?
15. Tell what is wrong with each of the following:

    (a)
    ```
    int n = 44;
    int* p = &n;
    ++(*p);
    int m = p;
    ```

(b)
```
int* p = new int ;
*p = 44;
int* q = p;
delete p;
```

(c)
```
int* p = new int;
*p = 44
int* q = new int ;
p = q;
```

(d)
```
int n = new int; n = 44;
```

16. Trace the following code, showing each value of each variable:
```
int a[] = {22, 33, 44, 55, 66, 77};
int* p = &a[3]; // assume that p gets the value 0x3fffcbc here
int n = *p; ++(*p);
++p;
int* q = &a[5];
*(--q) = 88;
p -= 3;
n = q - p;
```

17. Draw pictures to show the effect of the following code:
```
double* p = new double;
*p = 3.141592653589793;
short* q = new short[5];
*q = 44;
*(q+4) = 88;
```

18. Draw pictures to show the effects of the following statements:
```
string* p = new string("ABCDEFG");
string s = *p; // s is a copy of *p
string& r = *p; // r is a synonym for *p
string* q = &s; // q points to s
r[5] = '?';
q->erase(3, 2);
s[1] = '!';
p->replace(2, 1,"$=$");
```

19. Write declarations for each of the following
    (a) A pointer to a `char`.
    (b) A C-string which can have up to 19 characters.
    (c) A pointer to a C-string.
    (d) A `string` object.
    (e) A pointer to a `string` object.
    (f) A static array of 8 `string` objects.
    (g) A dynamic array of 8 `string` objects.

20. Write statements for each of the following:
    (a) Initialize a pointer `p` to point to a `Person` object `x`.
    (b) Assign the `Person` object `y` to the object to which `p` points.

(c) Initialize a pointer `q` to the address of the pointer `p`.

(d) Initialize a reference `r` to the `Person` object `x`.

21. Write and run a program that declares the following objects and then prints their addresses: a `bool`, a `short`, an `int`, a `float`, a `double`, a `string` of 5 characters, and an array of 5 `floats`:

    (a) using static allocation;

    (b) using dynamic allocation.

22. Implement the following function for the Purse class:

```
int f(Purse* p, Purse* q);
// Returns 1 if p and q point to the same purse.  Returns 0 if p
// and q point to different purses which have the same contents.
// Returns -1 if the two purses have different contents.
```

23. Implement the following function:

```
bool same(Person& x, Person& y);
// Returns true iff x and y are the same person.
```

24. Implement the following function from the ¡cstring¿ library for C-strings:

```
int strlen(const char* s);
// Returns the number of non-null characters in the C-string s.
```

25. Implement the following function from the <cstring> library for C-strings:

```
char* strcat(char* s1, const char* s2);
// Appends a copy of the C-string s2 to s1, and returns s1.
```

26. Implement the following function from the <cstring> library for C-strings:

```
char* strcpy(char* s1, const char* s2);
// Copies the non-null characters of s2 into s1, and returns s1.
```

27. Implement the following function from the <cstring> library for C-strings:

```
int strcmp(const char* s1, const char* s2);
// Compares s1 and s2 lexicographically.  Returns a negative
// integer is s1 < s2, a positive integer if s1 > s2, and 0
// if the two strings have the same value.
```

28. Implement the following function from the <cstring> library for C-strings:

```
char* strstr(const char* s1, const char* s2);
// Searches s1 for the substring s2.  If found, its address in s1
// is returned.  Otherwise, 0 is returned.
```