part II c++

chapter 11:

To support the principles of object-oriented programming, all OOP languages have three traits in common: encapsulation, polymorphism, and inheritance. Let's examine each.

כדי לתמוך בעקרונות של תכנות מונחה עצמים, לכל שפות OOP שלוש תכונות משותפות: אנקפסולציה, פולימורפיזם וירושה. בואו נבחן כל אחד.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.

הוא המנגנון הקושר את הקוד והנתונים שהוא מפעיל, ושומר על שניהם מפני הפרעות מבחוץ ושימוש לרעה. בשפה מונחה עצמים, קוד ונתונים עשויים להיות משולבים בצורה כזו שנוצרת "קופסה שחורה" עצמאית. כאשר קוד ונתונים מקושרים יחד בצורה זו, נוצר אובייקט. במילים אחרות, אובייקט הוא המכשיר התומך באנקפסולציה

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification.

ירושה היא התהליך שבו אובייקט אחד יכול לרכוש את המאפיינים של אובייקט אחר. זה חשוב כי זה תומך במושג הסיווג. אם אתה חושב על זה, רוב הידע נעשה לניהול על ידי סיווגים היררכיים

A Sample C++ Program

using namespace std;

This tells the compiler to use the std namespace. Namespaces are a recent addition

Copy string (function) C++. A namespace creates a declarative region in which various program elements can be placed.

זה אומר למהדר להשתמש במרחב השמות std. מרחבי שמות הם תוספת

לאחרונה
ל-C++. מרחב שמות יוצר אזור הצהרתי שבו ניתן למקם רכיבי תוכנית שונים.
don't need a namespace statement because the C library functions are also available in the default, global namespace.
אינן זקוקות להצהרת מרחב שמות מכיוון שפונקציות ספריית C זמינות גם ברירת המחדל של מרחב השמות הגלובלי.

Now examine the following line.
int main()
Notice that the parameter list in main( ) is empty. In C++, this indicates that main( ) has no parameters. This differs from C. In C, a function that has no parameters must use void in its parameter list, as shown here:
int main(void)
This was the way main( ) was declared in the programs in Part One. However, in C++, the use of void is redundant and unnecessary.

<< has an expanded role
It is still the left shift operator, but when it is used as shown in this example, it is also an output operator. The word cout is an identifier that is linked to the screen.
זה גם אופרטור פלט. המילה cout היא מזהה המקושר למסך.
You can use cout and the << to output any of the built-in data types, as well as strings of characters.
אתה יכול להשתמש ב-cout וב-<< כדי להוציא כל אחד מסוגי הנתונים המובנים, גם כן כמחרוזות של דמויות.

cin >> i;
In C++, the **>>** operator still retains its right shift meaning. However, when used as shown, it also is C++'s *input operator*. This statement causes **i** to be given a value read from the keyboard. The identifier **cin** refers to the standard input device, which
is usually the keyboard. In general, you can use **cin >>** to input a variable of any of the basic data types plus strings.
הצהרה זו גורמת ל-i לקבל ערך שנקרא מהמקלדת. המזהה cin מתייחס להתקן הקלט הסטנדרטי, אשר
היא בדרך כלל המקלדת.

```
return 0;
```
This causes zero to be returned to the calling process (which is usually the operating system).
vs
Abnormal program termination should be signaled by returning a nonzero value. You may also use the values EXIT_SUCCESS and EXIT_ FAILURE if you like.

A Closer Look at the I/O Operators

```
int main() {
	float f;
	char str[80];
	double d;
	cout << "Enter two floating point numbers: ";
	cin >> f >> d;
	cout << "Enter a string: ";
	cin >> str;
	cout << f << " " << d << " " << str;
return 0
```

When you run this program, try entering This is a test. when prompted for the string. When the program redisplays the information you entered, only the word "This" will be displayed. The rest of the string is not shown because the >> operator stops reading input when the first white-space character is encountered. Thus, "is a test" is

כאשר אתה מפעיל תוכנית זו, נסה להזין This is a test. כאשר תתבקש להזין את המחרוזת. כאשר התוכנית מציגה מחדש את המידע שהזנת, רק המילה "זה" תוצג. שאר המחרוזת לא מוצגת מכיוון שהאופרטור >> מפסיק לקרוא קלט כאשר נתקלים בתו הראשון של הרווח הלבן. לפיכך, "הוא מבחן" הוא

The C++ I/O operators recognize the entire set of backslash character constants described
cout << "A\tB\tC"; vs cout << f << " " << d << " " << str;
This statement outputs the letters A, B, and C, separated by tabs.

You cannot declare a variable in a block after an "action" statement

has occurred. For example, in C89, this fragment is incorrect:
In C++ (and C99) you may declare local variables at any point within a block

```
/* Incorrect in C89. OK in C++. */
      int f()
      {
int i; i = 10;
        int j;  /* won't compile as a C program */
        j = i*2;
return j; }
```

Whether you declare all variables at the start of a block or at the point of first use is completely up to you. Since much of the philosophy behind C++ is the encapsulation of code and data, it makes sense that you can declare variables close to where they are used instead of just at the beginning of the block. In the preceding example, the declarations are separated simply for illustration, but it is easy to imagine more complex examples in which this feature of C++ is more valuable.
Declaring variables close to where they are used can help you avoid accidental side effects. However, the greatest benefit of declaring variables at the point of first use is gained in large functions. Frankly, in short functions (like many of the examples in this book), there is little reason not to simply declare variables at the start of a function.
אם אתה מצהיר על כל המשתנים בתחילת הבלוק או בנקודת השימוש הראשון זה לגמרי תלוי בך. מכיוון שחלק גדול מהפילוסופיה שמאחורי C++ היא עטיפה של קוד ונתונים, הגיוני שתוכלו להצהיר על משתנים קרובים למקום שבו הם משמשים במקום רק בתחילת הבלוק. בדוגמה הקודמת, ההצהרות מופרדות רק לשם המחשה, אבל קל לדמיין דוגמאות מורכבות יותר שבהן תכונה זו של C++ היא בעלת ערך רב יותר.
הצהרה על משתנים קרובים למקום השימוש בהם יכולה לעזור לך להימנע מתופעות לוואי מקריות. עם זאת, התועלת הגדולה ביותר של הצהרת משתנים בנקודת השימוש הראשון מושגת בפונקציות גדולות. למען האמת, בפונקציות קצרות (כמו רבות מהדוגמאות בספר זה), אין סיבה מועטה פשוט לא להכריז על משתנים בתחילת פונקציה

However, when applied properly, especially in large functions, declaring variables at the point of their first use can help you create bug-free programs more easily.

No Default to int

Introducing C++ Classes

היכרות עם שיעורי C++

This section introduces C++'s most important feature: the class. In C++, to create an object, you first must define its general form by using the keyword **class**. A **class** is similar syntactically to a structure. Here is an example. The following class defines a type called **stack**, which will be used to create a stack:

היכרות עם שיעורי C++
חלק זה מציג את התכונה החשובה ביותר של C++: המחלקה. ב-C++, כדי ליצור אובייקט, תחילה עליך להגדיר את הצורה הכללית שלו באמצעות מחלקת מילות המפתח. מחלקה דומה מבחינה תחבירית למבנה. הנה דוגמא. המחלקה הבאה מגדירה סוג שנקרא מחסנית, אשר ישמש ליצירת מחסנית:

```
#define SIZE 100
/ This creates the class stack.
  class stack {
    int stck[SIZE];
    int tos;
  public:
    void init();
    void push(int i);
    int pop();
};
```

Once you have defined a class, you can create an object of that type by using the class name. In essence, the class name becomes a new data type specifier. For example, this creates an object called mystack of type stack:

לאחר שהגדרת מחלקה, תוכל ליצור אובייקט מסוג זה באמצעות שם המחלקה. בעצם, שם המחלקה הופך למפרט סוג נתונים חדש. לדוגמה, זה יוצר אובייקט בשם mystack מסוג stack

```
stack mystack;
```

When you declare an object of a class, you are creating an instance: of that class. In this case, mystack is an instance of stack.
כאשר אתה מכריז על אובייקט של מחלקה, אתה יוצר מופע של מחלקה זו. במקרה זה, mystack הוא מופע של מחסנית.

In C++, class creates a new data type that may be used to create

objects of that type. Therefore, an object is an instance of a class in just the same way that some other variable is an instance of the int data type,

ב-C++, מחלקה יוצרת סוג נתונים חדש שעשוי לשמש ליצירת אובייקטים מסוג זה. לכן, אובייקט הוא מופע של מחלקה בדיוק באותו אופן שבו משתנה אחר הוא מופע מסוג הנתונים int,

logical abstraction, while an object is real. (That is, an object exists inside the memory of the computer.)
The general form of a simple class declaration is:

```
class class-name {
private data and functions
public:
public data and functions } object name list;
```

Of course, the object name list may be empty.
Inside the declaration of stack, member functions were identified using their prototypes.

הפשטה לוגית, בעוד אובייקט הוא אמיתי. (כלומר, עצם קיים בתוך הזיכרון של המחשב.) כמובן שרשימת שמות האובייקט עשויה להיות ריקה.
בתוך ההצהרה של מחסנית, זוהו פונקציות איברים באמצעות שלהם אבות טיפוס.

In C++, all functions must be prototyped. Prototypes are not optional. The prototype for a member function within a class definition serves as that function's prototype in general.
When it comes time to actually code a function that is the member of a class, you must tell the compiler which class the function belongs to by qualifying its name with the name of the class of which it is a member. For example **push( )** function:

```
  void stack::push(int i)
  {
    if(tos==SIZE) {
      cout << "Stack is full.\n";
      return;
    }
    stck[tos] = i;
    tos++;
}
```

The :: is called the scope resolution operator.
ה-:: נקרא אופרטור רזולוציית ההיקף.

Essentially, it tells the compiler that this version of push( ) belongs to the stack class or, put differently, that this push( ) is in stack's scope. In C++, several different classes can use the same function name. The compiler knows which function belongs to which class because of the scope resolution operator ::

בעיקרו של דבר, זה אומר למהדר שגרסה זו של push( ) שייכת למחלקת המחסנית או, במילים אחרות, שה-push( ) זה נמצא בהיקף של מחסנית. ב-C++, מספר מחלקות שונות יכולות להשתמש באותו שם פונקציה. המהדר יודע איזו פונקציה שייכת לאיזו מחלקה בגלל האופרטור רזולוציית scope/ היקף

When you refer to a member of a class from a piece of code that is not part of the class, you must always do so in conjunction with an object of that class. To do so, use the object's name, followed by the dot operator, followed by the name of the member. This rule applies whether you are accessing a data member or a function member. For example, this calls init( ) for object stack1.

כאשר אתה מתייחס לחבר מחלקה מקטע קוד שאינו חלק מהמחלקה, עליך לעשות זאת תמיד בשילוב עם אובייקט של אותה מחלקה. לשם כך, השתמש בשם האובייקט, ואחריו באופרטור הנקודה, ואחריו שם החבר

```
stack stack1, stack2;
stack1.init();
```

there are example stack program at page 273

One last point:

Recall that the private members of an object are accessible only by functions that are members of that object.

For example, a statement like:

```
stack1.tos = 0; // Error, tos is private.
```

could not be in the main( ) function of the previous program because tos is private.

(אתה יכול גם ליצור אובייקטים כאשר המחלקה מוגדרת על ידי הצבת שמותיהם אחרי הסוגר המתולתל הסוגר, בדיוק כמו שאתה עושה עם מבנה.)

Function Overloading

One way that C++ achieves polymorphism is through the use of function overloading. In C++, two or more functions can share the same name as long as their parameter declarations are different. In this situation, the functions that share the same name are said to be overloaded, and the process is referred to as function overloading.

אחת הדרכים שבהן C++ משיג פולימורפיזם היא באמצעות שימוש בעומס יתר של פונקציות. ב-C++, שתי פונקציות או יותר יכולות לחלוק את אותו השם כל עוד הצהרות הפרמטר שלהן שונות. במצב זה, אומרים שהפונקציות שחולקות את אותו השם עומסות יתר על המידה, והתהליך מכונה עומס יתר של פונקציות.

כדי לראות מדוע חשובה העמסת יתר של פונקציות,

```
#include <iostream>
  using namespace std;
  // abs is overloaded three ways
  int abs(int i);
  double abs(double d);
  long abs(long l);
int main() {
    cout << abs(-10) << "\n";
    cout << abs(-11.0) << "\n";
    cout << abs(-9L) << "\n";
    return 0;
}

long abs(long l)
  {
    cout << "Using long abs()\n";
    return l<0 ? -l : l;
  }
```

. הדוגמה הזו טריוויאלית למדי, אבל אם תרחיב את הרעיון, תוכל לראות כיצד פולימורפיזם יכול לעזור לך לנהל תוכניות מורכבות מאוד.

In general, to overload a function, simply declare different versions of it. The compiler takes care of the rest

You must observe one important restriction when overloading a function: the type and/or number of the parameters of each overloaded function must differ. It is not sufficient for two functions to differ only in their return types. They must differ in the types or

number of their parameters. (Return types do not provide sufficient information in all cases for the compiler to decide which function to use.)

באופן כללי, כדי להעמיס על פונקציה, פשוט הכריז על גרסאות שונות שלה. המהדר דואג לשאר. עליך להקפיד על הגבלה חשובה אחת בעת עומס יתר על פונקציה: סוג ו/או מספר הפרמטרים של כל פונקציה עמוסה חייבים להיות שונים. לא מספיק ששתי פונקציות יהיו שונות רק בסוגי ההחזר שלהן. הם חייבים להיות שונים בסוגים או במספר הפרמטרים שלהם. (סוגי החזרה אינם מספקים מידע מספק בכל המקרים כדי שהמהדר יחליט באיזו פונקציה להשתמש.)

```
void stradd(char *s1, char *s2);
void stradd(char *s1, int i);
void stradd(char *s1, char *s2 ,int i);type or number parameter,
(parameters=s2) (operator=*)
```

Operator Overloading

Polymorphism is also achieved in C++ through operator overloading. As you know, in C++, it is possible to use the << and >> operators to perform console I/O operations. They can perform these extra operations because in the <iostream> header, these operators are overloaded. When an operator is overloaded, it takes on an additional meaning relative to a certain class. However, it still retains all of its old meanings.

עומס יתר על המפעיל

פולימורפיזם מושג גם ב-C++ באמצעות עומס יתר של מפעיל. כידוע, ב-C++, אפשר להשתמש באופרטורים << ו-> כדי לבצע פעולות קלט/פלט של קונסולה. הם יכולים לבצע את הפעולות הנוספות הללו מכיוון שבכותרת <iostream>, האופרטורים הללו עמוסים יתר על המידה. כאשר מפעיל עומס יתר על המידה, הוא מקבל משמעות נוספת ביחס למחלקה מסוימת. עם זאת, הוא עדיין שומר על כל המשמעויות הישנות שלו.

operator overloading is, in practice, somewhat more complex than function overloading,

examples are deferred unti, look at Chapter 14.

Inheritance

As stated earlier in this chapter, inheritance is one of the major traits of an object- oriented programming language. In C++, inheritance is

supported by allowing one class to incorporate another class into its declaration. Inheritance allows a hierarchy of classes to be built, moving from most general to most specific. The process involves first defining a *base class*, which defines those qualities common to all objects to be derived from the base. The base class represents the most general description. The classes derived from the base are usually referred to as *derived classes*. A derived class includes all features of the generic base class and then adds qualities specific to the derived class. To demonstrate how this works, the next example creates classes that categorize different types of buildings.
To begin, the **building** class is declared, as shown here. It will serve as the base for two derived classes.

ירושה
כפי שצוין מוקדם יותר בפרק זה, תורשה היא אחת התכונות העיקריות של שפת תכנות מונחה עצמים. ב-C++, ירושה נתמכת על ידי מתן אפשרות למחלקה אחת לשלב מחלקה אחרת בהצהרה שלה. הירושה מאפשרת לבנות היררכיה של מחלקות, העוברות מהכלליות ביותר לספציפיות ביותר. התהליך כולל תחילה הגדרת מחלקה בסיס, המגדירה את התכונות המשותפות לכל האובייקטים שייגזרו מהבסיס. מחלקת הבסיס מייצגת את התיאור הכללי ביותר. המחלקות הנגזרות מהבסיס מכונות בדרך כלל כיתות נגזרות. מחלקה נגזרת כוללת את כל התכונות של מחלקת הבסיס הגנרית ולאחר מכן מוסיפה איכויות ספציפיות למחלקה הנגזרת. כדי להדגים איך זה עובד, הדוגמה הבאה יוצרת כיתות שמסווגות סוגים שונים של מבנים.
כדי להתחיל, מעמד הבניין מוכרז, כפי שמוצג כאן. זה ישמש כבסיס לשתי כיתות נגזרות.

You can now use this broad definition of a building to create derived classes that describe specific types of buildings. For example, here is a derived class called house:
כעת תוכל להשתמש בהגדרה הרחבה הזו של בניין כדי ליצור מחלקות נגזרות המתארות סוגים ספציפיים של מבנים. לדוגמה, הנה מחלקה נגזרת הנקראת בית:
all inherited classes
will use public. Using public means that all of the public members of the base class will become public members of the derived class.
כל המחלקות שעברו בירושה
ישתמש בציבור. שימוש בציבור פירושו שכל חברי הציבור במעמד הבסיס יהפכו לחברים ציבוריים במעמד הנגזר
Therefore, the public members of the class building become public members of the derived class house and are available to the member functions of house just as if they had been declared inside house. However, house's member functions do not have access to the

private elements of building. This is an important point. Even though house inherits building, it has access only to the public members of building.

לכן, חברי הציבור של בניין הכיתה הופכים לחברי ציבור של בית הכיתה הנגזר וזמינים לתפקידי חברי הבית ממש כאילו הוכרזו בתוך הבית. עם זאת, לפונקציות החברים בבית אין גישה לאלמנטים הפרטיים של הבניין. זו נקודה חשובה. למרות שהבית יורש בניין, יש לו גישה רק לחברי הציבור של הבניין.

In this way, inheritance does not circumvent the principles of encapsulation necessary to OOP.

Here is a program illustrating inheritance. It creates two derived classes of building using inheritance; one is house, the other, school.

לפניכם תוכנית הממחישה ירושה. זה יוצר שני מחלקות נגזרות של בנייה תוך שימוש בירושה; האחד הוא בית, השני, בית ספר.

page 280

As this program shows, the major advantage of inheritance is that you can create a general classification that can be incorporated into more specific ones. In this way, each object can precisely represent its own subclass.

When writing about C++, the terms base and derived are generally used to describe the inheritance relationship. However, the terms parent and child are also used. You may also see the terms superclass and subclass.

Aside from providing the advantages of hierarchical classification, inheritance also provides support for run-time polymorphism through the mechanism of virtual functions. (Refer to Chapter 16 for details.)

כפי שמראה תוכנית זו, היתרון העיקרי של ירושה הוא שאתה יכול ליצור סיווג כללי שניתן לשלב בסיווג ספציפי יותר. בדרך זו, כל אובייקט יכול לייצג במדויק תת-מחלקה משלו.

כאשר כותבים על C++, המונחים בסיס ונגזר משמשים בדרך כלל לתיאור יחסי הירושה. עם זאת, נעשה שימוש גם במונחים הורה וילד. ייתכן שתראה גם את המונחים מחלקה-על ותת-מחלקה.

מלבד מתן היתרונות של סיווג היררכי, תורשה מספקת גם תמיכה לפולימורפיזם בזמן ריצה באמצעות מנגנון של פונקציות וירטואליות. (עיין בפרק 16 לפרטים.)

Constructors and Destructors

It is very common for some part of an object to require initialization before it can be used. For example, think back to the stack class developed earlier in this chapter. Before the stack could be used, tos

had to be set to zero. This was performed by using the function init( ). Because the requirement for initialization is so common, C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor function.

```
/ This creates the class stack.
  class stack {
    int stck[SIZE];
    int tos;
  public:
    stack();  // constructor
    void push(int i);
    int pop();
};
```

Notice that the constructor **stack( )** has no return type specified. In C++, constructors cannot return values and, thus, have no return type.

The **stack( )** constructor is coded like this:

```
// stack's constructor 'must to be inside to stack'
  stack::stack()
  {
tos = 0;
    cout << "Stack Initialized\n";
  }
```

the constructor will simply perform various initializations.

פשוט יבצעו אתחולים שונים

An object's constructor is automatically called when the object is created. This means that it is called when the object's declaration is executed. If you are accustomed to thinking of a declaration statement as being passive, this is not the case for C++. In C++, a declaration statement is a statement that is executed.

הבנאי של אובייקט נקרא אוטומטית כאשר האובייקט נוצר. זה אומר שהוא נקרא כאשר ההצהרה של האובייקט מבוצעת. אם אתה רגיל לחשוב על הצהרת הצהרה כפאסיבית, זה לא המקרה עבור C++. ב-C++, משפט הכרזה הוא משפט שמתבצע.

ההבחנה הזו היא לא רק אקדמית. הקוד שמבוצע לבניית אובייקט עשוי להיות משמעותי למדי.

There are many reasons why a destructor may be needed. For example, an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened. In C++, it is the destructor that handles deactivation events. The destructor has the same name as the constructor, but it is preceded by a ~. For example, here is the stack class and its constructor and destructor. (Keep in mind that the stack class does not require a destructor; the one shown here is just for illustration.)

ישנן סיבות רבות מדוע ייתכן שיהיה צורך במשמיד. לדוגמה, ייתכן שאובייקט יצטרך להקצות זיכרון שהוא הקצה בעבר או שהוא עשוי להצטרך לסגור קובץ שהוא פתח. ב-C++, ה-Destructor הוא שמטפל באירועי השבתה. להרס יש אותו שם כמו ה-בנאי, אבל לפניו הוא ~. לדוגמה, הנה מחלקת המחסנית והקונסטרוקטור וההרס שלה. (זכור שמחלקת המחסנית אינה דורשת הרס; זה שמוצג כאן הוא רק להמחשה.)

```
// This creates the class stack.
  class stack {
    int stck[SIZE];
    int tos;
  public:
    stack();  // constructor
    ~stack(); // destructor
    void push(int i);
    int pop();
};
// stack's constructor
  stack::stack()
  {
tos = 0;
    cout << "Stack Initialized\n";
  }
  // stack's destructor
  stack::~stack()
  {
    cout << "Stack Destroyed\n";
  }
```

The General Form of a C++ Program

Although individual styles will differ, most C++ programs will have this

general form:

#includes
base-class declarations
derived class declarations nonmember function prototypes int main( )
{
//... }
nonmember function definitions
In most large projects, all **class** declarations will be put into a header file and included with each module. But the general organization of a program remains the same.
The remaining chapters in this section examine in greater detail the features
discussed in this chapter, as well as all other aspects of C++.

chapter 12

Classes

Classes are created using the keyword class. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an instance of a class.
A class declaration is similar syntactically to a structure. In Chapter 11, a simplified general form of a class declaration was shown. Here is the entire general form of a class declaration that does not inherit any other class.

הכיתות נוצרות באמצעות מילת המפתח class. הצהרת מחלקה מגדירה סוג חדש המקשר קוד ונתונים. סוג חדש זה משמש לאחר מכן להכרזה על אובייקטים מאותה מחלקה. לפיכך, מחלקה היא הפשטה לוגית, אך לאובייקט יש קיום פיזי. במילים אחרות, אובייקט הוא מופע של מחלקה.
הצהרת מחלקה דומה מבחינה תחבירית למבנה. בפרק 11 הוצגה צורה כללית פשוטה של הצהרת מעמדות. הנה כל הצורה הכללית של הצהרת מעמדות שאינה יורשת אף מחלקה אחרת.

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class.

כברירת מחדל, פונקציות ונתונים המוצהרים בתוך מחלקה הם פרטיים לאותה מחלקה וניתן לגשת אליהם רק חברים אחרים במחלקה.

The public access specifier allows functions or data to be accessible to other parts of your program. The protected access specifier is needed only when inheritance is involved (see Chapter 15). Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

מפרט הגישה הציבורי מאפשר לפונקציות או נתונים להיות נגישים לחלקים אחרים של התוכנית שלך. יש צורך במפרט הגישה המוגנת רק כאשר מדובר בירושה (ראה פרק 15). לאחר שנעשה שימוש במפרט גישה, הוא יישאר בתוקף עד להיתקל במפרט גישה אחר או להגיע לסוף הצהרת המחלק

Functions that are declared within a class are called member functions.
Member functions may access any element of the class of which they are a part.

פונקציות המוצהרות בתוך מחלקה נקראות פונקציות איבר. פונקציות חבר עשויות לגשת לכל רכיב במחלקה שהן חלק ממנה.

This includes all private elements. Variables that are elements of a class are called member variables or data members. (The term instance variable is also used.) Collectively, any element of a class can be referred to as a member of that class.

There are a few restrictions
that apply to class members. A non-static member variable cannot have an initializer.
No member can be an object of the class that is being declared. (Although a member can be a pointer to the class that is being declared.)
No member can be declared as auto, extern, or register.

יש כמה הגבלות
החלים על חברי הכיתה. למשתנה חבר לא סטטי לא יכול להיות אתחול.
אף חבר לא יכול להיות אובייקט של המחלקה המוצהרת. (למרות שחבר יכול להיות מצביע למחלקה המוצהרת.) אין להכריז על איבר כאוטומטי, חיצוני או רישום.

(The term instance variable is also used.) Collectively, any element of a class can be referred to as a member of that class.

```
class employee { //A class declaration defines a new type that links
code and data.
  char name[80]; // private by default,
public:
  void putname(char *n); // these are public
  void getname(char *n);
private:
  double wage; // now, private again, Variables that are elements of a
class are called member variables or data members. משתנים שהם
אלמנטים של מחלקה נקראים משתני איברים או איברי נתונים.

public://The public access specifier allows functions or data to be
accessible to other parts of your program like main().
  void putwage(double w); // back to public, other members of the
class
                                        //Functions that are declared within a class
are called member functions.
  double getwage();//other members of the class
};


Member functions may access any element of the class of which they
are a part.
                    פונקציות חבר עשויות לגשת לכל רכיב במחלקה שהן חלק ממנה.
void employee::getname(char *n)
{
  strcpy(n, name);
}
void employee::putwage(double w)
{
wage = w; }member can access to private element


int main(){
The public access specifier allows functions or data to be accessible
to other parts of your program.
  employee emy //a class is a logical abstraction, but an object has
physical existence
```

```
  char name[80];
  ted.putname("Ted Jones");
  ted.putwage(75000);

}
```

vs

```
#include <iostream>
   using namespace std;
   class myclass {
   public:
     int i, j, k; // accessible to entire program
   };
int main() {
myclass a, b;
     a.i = 100; // access to i, j, and k is OK
     a.j = 4;
     a.k = a.i * a.j;
     b.k = 12; // remember, a.k and b.k are different
     cout << a.k << " " << b.k;
return 0; }
```

page 294.

Structures and Classes Are Related

Structures are part of the C subset and were inherited from the C language. As you have seen, a class is syntactically similar to a struct. But the relationship between a class and a struct is closer than you may at first think. In C++, the role of the structure was expanded, making it an alternative way to specify a class. In fact, the only difference between a class and a struct is that by default all members are public in a struct and private in a class. In all other respects, structures and classes are equivalent. That is,
in C++, a structure defines a class type. For example, consider this short program, which uses a structure to declare a class that controls access to a strin

מבנים הם חלק מתת-קבוצת C ועברו בירושה משפת C. כפי שראית,
מחלקה דומה מבחינה תחבירית למבנה. אבל הקשר בין כיתה למבנה קרוב
יותר ממה שאתה עשוי לחשוב בהתחלה. ב-C++, תפקיד המבנה הורחב,
מה שהפך אותו לדרך חלופית לציון מחלקה. למעשה, ההבדל היחיד בין

מחלקה למבנה הוא שבברירת מחדל כל החברים הם ציבוריים במבנה
ופרטים במחלקה. מכל הבחינות האחרות, מבנים וכיתות שווים. זה,
ב-C++, מבנה מגדיר סוג מחלקה. לדוגמה, שקול את התוכנית הקצרה הזו,
המשתמשת במבנה כדי להכריז על מחלקה השולטת בגישה למחרוזת

strncat: Append characters from string (function)
strcpy: Copy string (function)
memcpy: Copy block of memory

```
struct mystr {
    void buildstr(char *s); // public
    void showstr();
  private: // now go private
    char str[255];//struct members are public by default
};
  void mystr::buildstr(char *s)
  {
    if(!*s) *str = '\0'; // initialize string
    else strcat(str, s);
  }
  void mystr::showstr()
  {
    cout << str << "\n";
  }
int main(){
mystr s; //object declared
 const char *p="Hello word"; pointer Array Initializations
  or
  char str[] = “Hello word!”; // Unsized Array Initializations

 s.buildstr(str or p);

 s.showstr();
}
```

vs
This program displays the string **Hello there!**.

The class **mystr** could be rewritten by using **class** as shown here:

```
  class mystr {
    char str[255];// class members private by default
  public:
```

```
    void buildstr(char *s); // public
    void showstr();
};
```

page 296

Unions and Classes Are Related

Like a structure, a **union** may also be used to define a class. In C++, **union**s may contain both member functions and variables. They may also include constructors and destructors. A **union** in C++ retains all of its C-like features, the most important being that all data elements share the same location in memory. Like the structure, **union** members are public by default and are fully compatible with C. In the next share the same location in memory.
example, a **union** is used to swap the bytes that make up an **unsigned short** integer.
(This example assumes that short integers are 2 bytes long.)

איגודים וחוגים קשורים זה לזה

כמו מבנה, איחוד עשוי לשמש גם להגדרת מחלקה. ב-C++, איגודים עשויים להכיל גם פונקציות חבר וגם משתנים. הם עשויים לכלול גם בנאים והרסים. איחוד ב-C++ שומר על כל התכונות דמויות ה-C שלו, והחשוב ביותר הוא שכל רכיבי הנתונים חולקים את אותו מיקום בזיכרון. כמו המבנה, חברי האיגוד הם ציבוריים כברירת מחדל והם תואמים לחלוטין ל-C. בדוגמה הבאה, נעשה שימוש באיחוד כדי להחליף את הבתים המרכיבים מספר שלם קצר ללא סימן. (דוגמה זו מניחה שאורכם של מספרים שלמים קצרים הוא 2 בתים.)

```
// share the same location in memory this program
#include <iostream>
  using namespace std;
  union swap_byte {//A union declaration defines a new type that links code and data.
    void swap();
    void set_byte(unsigned short i);
    void show_word();
    unsigned short u;//union members are public by default
    unsigned char c[2];
  };
  void swap_byte::swap()
  {
   unsigned char t;
   t = c[0];
  c[0] = c[1];
```

```
   c[1] = t;
 }
     void swap_byte::show_word()
     {
cout << u; }
     void swap_byte::set_byte(unsigned short i)
     {
u = i; }
int main() {
swap_byte b;
       b.set_byte(49034);
       b.swap();
       b.show_word();
return 0; }
```

another example:

```
union swap_byte {//
    void swap();
    void set_byte(unsigned short i);
    void show_word();
    unsigned short u;//union members are public by default (u=16 bit)
    unsigned char c[2];// arr 8 bits, it will access to place of h variable
at memory
  };                  //  and print, union share same location memory.

     void swap_byte::show_word()
     {
    cout << c[0]<<"\n";
    cout << c<<"\n"; }
     void swap_byte::set_byte(unsigned short i)
     {    u = i; }
int main() {
       swap_byte b;
       b.set_byte(56);//
       b.show_word();
return 0; }
```

Like a structure, a union declaration in C++ defines a special type of class.
This means that the principle of encapsulation is preserved.
There are several restrictions that must be observed when you use C++ unions.

מגדירה סוג מיוחד של מחלקה ++C-כמו מבנה, הצהרת איחוד ב.
המשמעות היא שעיקרון האנקפסולציה נשמר.
ישנן מספר הגבלות שיש להקפיד עליהן כאשר אתה משתמש באיגודי C++.

First, a union cannot inherit any other classes of any type.

Further, a union cannot be a base class.

A union cannot have virtual member functions.
(Virtual functions are discussed in Chapter 17.)

No static variables can be members of a union.

A reference member cannot be used.

A union cannot have as a member any object that overloads the = operator.

Finally, no object can be a member of a union if the object has an explicit constructor or destructor function.

As with struct, the term POD is also commonly applied to unions that do not contain member functions, constructors, or destructors.

Anonymous Unions

There is a special type of **union** in C++ called an *anonymous union*. An anonymous union does not include a type name, and no objects of the union can be declared.
Instead, an anonymous union tells the compiler that its member variables are to share the same location. However, the variables themselves are referred to directly, without the normal dot operator syntax. For example, consider this program:

איגודים אנונימיים
יש סוג מיוחד של איחוד ב-C++ שנקרא איחוד אנונימי. איגוד אנונימי אינו כולל שם סוג, ולא ניתן להצהיר על אובייקטים של האיגוד.
במקום זאת, איגוד אנונימי אומר למהדר שהמשתנים החברים בו אמורים לחלוק את אותו מיקום. עם זאת, המשתנים עצמם מופנים ישירות, ללא תחביר אופרטור הנקודה הרגיל

```
int main() {

    // define anonymous union
```

```
    union {
      long l;
      double d;
      char s[4];
};
    // now, reference union elements directly
    l = 100000;
    cout << l << " ";
    d = 123.2342;
    cout << d << " ";
    strcpy(s, "hi");
    cout << s;
return 0;
}
```

anonymous union must not conflict with other identifiers known within the same scope.
Anonymous unions cannot contain private or protected elements.
Finally, global anonymous unions must be specified as static.

friend function

It is possible to grant a nonmember function access to the private members of a class by using a friend. A friend function has access to all private and protected members
of the class for which it is a friend. To declare a friend function, include its prototype within the class, preceding it with the keyword friend. Consider this program:

ניתן להעניק לפונקציה שאינה חברים גישה לחברים הפרטיים בכיתה באמצעות חבר. לפונקציית חבר יש גישה לכל החברים הפרטיים והמוגנים מהכיתה שעבורה הוא חבר. כדי להכריז על פונקציית חבר, כלול את אב הטיפוס שלה בתוך המחלקה, לפניה עם מילת המפתח חבר. שקול את התוכנית הזו:

keyword=friend and the name class

```
  class myclass {
    int a, b;
  public:
      sum is  nonmember function of class but if we using keyword
‘friend ‘
       in sum so the sum has access to all private and protected
members
```

```
of the class for which it is a friend.   access to members with name
class 'myclass x'
    friend int sum(myclass x);<-
  vs  set_ab is member friend so can access to private
   void set_ab(int i, int j);
};
   void myclass::set_ab(int i, int j)
   {
a = i;
b = j; }
   // Note: sum() is not a member function of any class.
   int sum(myclass x)
   {
    /* Because sum() is a friend of myclass, it can
       directly access a and b. */
    return x.a + x.b;
   }
int main() {
    myclass n;
    n.set_ab(3, 4);
    cout << sum(n);
    return 0;
}
```

Although there is nothing gained by making sum( ) a friend rather than a member function of myclass, there are some circumstances in which friend functions are quite valuable. First, friends can be useful when you are overloading certain types of operators (see Chapter 14). Second, friend functions make the creation of some types of I/O functions easier (see Chapter 18). The third reason that friend functions may be desirable is that in some cases, two or more classes may contain members that are interrelated relative to other parts of your program. Let's examine this third usage now.

שתי מחלקות או יותר עשויות להכיל חברים הקשורים זה בזה ביחס לחלקים אחרים של התוכנית שלך. הבה נבחן את השימוש השלישי הזה כעת.

friend function more example

so that no message is accidentally overwritten. Although you can create member functions in each class that return a value indicating whether a message is active,

this means additional overhead when the condition is checked (that is, two function calls, not just one). If the condition needs to be

checked frequently, this additional overhead may not be acceptable. However, using a function that is a friend of each class, it is possible to check the status of each object by calling only this one function. Thus, in situations like this, a friend function allows you to generate more efficient code. The following program illustrates this concept:
כך שאף הודעה לא תוחלף בטעות. למרות שאתה יכול ליצור פונקציות חבר בכל מחלקה שמחזירות ערך המציין אם הודעה פעילה,
המשמעות היא תקורה נוספת כאשר התנאי נבדק (כלומר, שתי קריאות פונקציה, לא רק אחת). אם יש צורך לבדוק את המצב לעתים קרובות, ייתכן שהתקורה הנוספת הזו לא תהיה מקובלת. עם זאת, באמצעות פונקציה שהיא חברה של כל מחלקה, ניתן לבדוק את המצב של כל אובייקט על ידי קריאה לפונקציה האחת הזו בלבד. לפיכך, במצבים כאלה, פונקציית חבר מאפשרת ליצור קוד יעיל יותר. התוכנית הבאה ממחישה את הרעיון הזה:

```
#include <iostream>
  using namespace std;
  const int IDLE = 0;
  const int INUSE = 1;
```

Notice that this program uses a forward declaration (also called a forward reference) for the class C2. This is necessary because the declaration of idle( ) inside C1 refers to C2 before it is declared.
שימו לב שתוכנית זו משתמשת בהצהרה קדימה (נקראת גם הפניה קדימה) עבור המחלקה C2. זה הכרחי מכיוון שההצהרה של idle( ) בתוך C1 מתייחסת ל-C2 לפני שהוא מוצהר. כדי ליצור הצהרה קדימה לכיתה, פשוט השתמש בטופס המוצג בתוכנית זו.
חבר של כיתה אחת עשוי להיות חבר בכיתה אחרת. לדוגמה, הנה התוכנית הקודמת נכתבה מחדש כך שה- idle( ) הוא חבר ב-C1:

```
  class C2;  // forward declaration
  class C1 {
    int status;  // IDLE if off, INUSE if on screen
    // ...
  public:
    void set_status(int state);
    friend int idle(C1 a, C2 b);
};
  class C2 {
    int status; // IDLE if off, INUSE if on screen
    // ...
  public:
    void set_status(int state);
    friend int idle(C1 a, C2 b);
};
```

```
void C1::set_status(int state)
  {
    status = state;
  }
  void C2::set_status(int state)
  {
    status = state;
  }
  int idle(C1 a, C2 b)
  {
    if(a.status || b.status) return 0;
    else return 1;
  }
int main() {
C1 x; C2 y;
    x.set_status(IDLE);
    y.set_status(IDLE);
    if(idle(x, y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    x.set_status(INUSE);
    if(idle(x, y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
return 0; }
```

Notice that this program uses a *forward declaration* (also called a *forward reference*) for the class **C2**. This is necessary because the declaration of **idle( )** inside **C1** refers to **C2** before it is declared. To create a forward declaration to a class, simply use the form shown in this program.

A **friend** of one class may be a member of another. For example, here is the preceding program rewritten so that **idle( )** is a member of **C1**:

שימו לב שתוכנית זו משתמשת בהצהרה קדימה (נקראת גם הפניה קדימה) עבור המחלקה C2. זה הכרחי מכיוון שההצהרה של idle( ) בתוך C1 מתייחסת ל-C2 לפני שהוא מוצהר. כדי ליצור הצהרה קדימה לכיתה, פשוט השתמש בטופס המוצג בתוכנית זו.

חבר של כיתה אחת עשוי להיות חבר בכיתה אחרת. לדוגמה, הנה התוכנית הקודמת נכתבה מחדש כך שה- idle( ) הוא חבר ב-C1:

or

```
class C1 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
```

```
  void set_status(int state);
  int idle(C2 b);  // now  a member of C1: it mean function idle() as
default  has access
};
class C2 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
 //because idle() member of c1 we declare as C1::idle and not as idle
  friend int C1::idle(C2 b);
};

// idle() is member of C1, but friend of C2
int C1::idle(C2 b)
{
  if(status || b.status) return 0;
  else return 1;
}

int main()
C1 x; C2 y;
     x.set_status(IDLE);
     y.set_status(IDLE);
    // obj x hold the value/pointer of variable of class C1
    // the compiler know it from C1 by declare C1::idle
     if(x.idle(y)) cout << "Screen can be used.\n";
     else cout << "In use.\n";
     x.set_status(INUSE);
     if(x.idle(y)) cout << "Screen can be used.\n";
     else cout << "In use.\n";
return 0; }
```

to sum up :
in addition Because idle( ) is a member of C1, it can access the status variable of objects of type C1 directly. Thus, only objects of type C2 need be passed to idle( ).

There are two important restrictions that apply to friend functions. First, a derived class does not inherit friend functions. Second, friend functions may not have a storage-class specifier. That is, they may not be declared as static or extern.

מכיוון ש- idle( ) הוא חבר ב-C1, הוא יכול לגשת ישירות למשתנה הסטטוס

של אובייקטים מסוג C1. לפיכך, רק אובייקטים מסוג C2 צריכים לעבור ל-idle( ).

ישנן שתי הגבלות חשובות החלות על פונקציות חבר. ראשית, מחלקה נגזרת אינה יורשת פונקציות חבר. שנית, ייתכן שלפונקציות חבר אין מפרט מחלקת אחסון. כלומר, אסור להכריז עליהם כסטטיים או חיצוניים.

Friend Classes

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class. For example,

In this example, class **Min** has access to the private variables **a** and **b** declared within the **TwoValues** class.
It is critical to understand that when one class is a **friend** of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the **friend** class.
Friend classes are seldom used. They are supported to allow certain special case situations to be handled.

אפשר שכיתה אחת תהיה חברה של כיתה אחרת. כאשר זה המקרה, למחלקת החברים ולכל פונקציות החברים שלה יש גישה לחברים הפרטיים המוגדרים בתוך המחלקה האחרת. לדוגמה,

בדוגמה זו, למחלקה Min יש גישה למשתנים הפרטיים a ו-b המוצהרים במחלקה TwoValues.
חשוב להבין שכאשר כיתה אחת היא חברה של אחרת, יש לה גישה רק לשמות המוגדרים בתוך הכיתה השנייה. זה לא יורש את המעמד השני.
באופן ספציפי, חברי הכיתה הראשונה לא הופכים לחברים בכיתת החברים.
שיעורי חברים משמשים לעתים רחוקות. הם נתמכים כדי לאפשר טיפול במצבי מקרים מיוחדים מסוימים.

```
// Using a friend class.
  #include <iostream>
  using namespace std;
  class TwoValues {
    int a;
```

```
    int b;
  public:
    TwoValues(int i, int j) { a = i; b = j; }
    friend class Min;
  };
class Min {

public:
    int min(TwoValues x);
};
  int Min::min(TwoValues x)
  {
    return x.a < x.b ? x.a : x.b;
  }
int main() {
    TwoValues ob(10, 20);
    Min m;
    cout << m.min(ob);
return 0; }
```

inline Functions

There is an important feature in C++, called an inline function, that is commonly used with classes. Since the rest of this chapter (and the rest of the book) will make heavy use of it, inline functions are examined here.

פונקציות מוטבעות

יש תכונה חשובה ב-C++, הנקראת פונקציה מוטבעת, שנמצאת בשימוש נפוץ עם מחלקות. מכיוון ששאר פרק זה (ושאר הספר) יעשה בו שימוש רב, נבדקות כאן פונקציות מוטבעות.

```
inline int max(int a, int b)
  {
    return a>b ? a : b;
  }
int main() {
    cout << max(10, 20);
    cout << " " << max(99, 88);
return 0; }
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

```
  #include <iostream>
```

```
  using namespace std;
int main() {
    cout << (10>20 ? 10 : 20);
    cout << " " << (99>88 ? 99 : 88);
return 0; }
```

However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to inline only very small functions. Further, it is also
a good idea to inline only those functions that will have significant impact on the performance of your program.
Like the register specifier, inline is actually just a request, not a command, to the compiler. The compiler can choose to ignore it. Also, some compilers may not inline

עם זאת, כאשר פונקציה מורחבת בשורה, אף אחת מהפעולות הללו לא מתרחשת. למרות שהרחבת קריאות פונקציות בתור יכולה לייצר זמני ריצה מהירים יותר, היא יכולה גם לגרום לגודל קוד גדול יותר בגלל קוד משוכפל. מסיבה זו, עדיף לשלב רק פונקציות קטנות מאוד. יתר על כן, זה גם כן רעיון טוב לשלב רק את הפונקציות שתהיה להן השפעה משמעותית על ביצועי התוכנית שלך.
כמו מפרט האוגר, ה-inline הוא למעשה רק בקשה, לא פקודה, למהדר. המהדר יכול לבחור להתעלם ממנו. כמו כן, ייתכן שחלק מהמהדרים אינם מוטבעים

note:
. כפי שאתם בוודאי יודעים, בכל פעם שפונקציה נקראת, נוצרת כמות משמעותית של תקורה על ידי מנגנון הקריאה והחזרה. בדרך כלל, ארגומנטים נדחפים אל המחסנית ואוגרים שונים נשמרים כאשר קוראים לפונקציה, ולאחר מכן משוחזרים כאשר הפונקציה חוזרת. הצרה היא שההוראות האלה לוקחות זמן.

Remember, if a function cannot be inlined, it will simply be called as a normal function.
Inline functions may be class member functions. For example, this is a perfectly valid C++ program:

```
#include <iostream>
  using namespace std;
  class myclass {
    int a, b;
```

```
  public:
    void init(int i, int j);
    void show();
};
  // Create an inline function.
  inline void myclass::init(int i, int j)
  {
a = i;
b = j; }
  // Create another inline function.
  inline void myclass::show()
  {
    cout << a << " " << b << "\n";
  }
int main() {
myclass x;
    x.init(10, 20);
    x.show();
return 0; }
```

Defining Inline Functions Within a Class

It is possible to define short functions completely within a class declaration. When a function is defined inside a class declaration, it is automatically made into an **inline** function (if possible). It is not necessary (but not an error) to precede its declaration with the **inline** keyword. For example, the preceding program is rewritten here with the definitions of **init( )** and **show( )** contained within the declaration of **myclass**:

הגדרת פונקציות מוטבעות בתוך מחלקה

ניתן להגדיר פונקציות קצרות לחלוטין בתוך הצהרת מחלקה. כאשר פונקציה מוגדרת בתוך הצהרת מחלקה, היא הופכת אוטומטית לפונקציה מוטבעת (אם אפשר). אין צורך (אך לא שגיאה) להקדים את ההצהרה שלו עם מילת המפתח המוטבעת. לדוגמה, התוכנית הקודמת נכתבת כאן מחדש עם ההגדרות של init( ) ו-show( ) הכלולות בהצהרה של myclass:

```
#include <iostream>
    using namespace std;
    class myclass {
      int a, b;
    public:
```

```
        // automatic inline
        void init(int i, int j) { a=i; b=j; }
        void show() { cout << a << " " << b << "\n"; }
};
int main() {
myclass x;
        x.init(10, 20);
        x.show();
return 0; }
```

Technically, the inlining of the show( ) function is of limited value because (in general) the amount of time the I/O statement will take far exceeds the overhead
of a function call. However, it is extremely common to see all short member functions defined inside their class in C++ programs. (In fact, it is rare to see short member functions defined outside their class declarations in professionally written C++ code.)
Constructor and destructor functions may also be inlined, either by .default, if defined within their class, or explicitly

מבחינה טכנית, הטבעה של הפונקציה show( ) היא בעלת ערך מוגבל מכיוון ש(באופן כללי) משך הזמן שהצהרת ה-I/O ייקח חורג בהרבה מהתקורה של קריאת פונקציה. עם זאת, נפוץ מאוד לראות את כל פונקציות האיברים הקצרות המוגדרות בתוך המחלקה שלהן בתוכניות C++. (למעשה, נדיר לראות פונקציות חבר קצרות המוגדרות מחוץ להצהרות המחלקות שלהן בקוד C++ שנכתב בצורה מקצועית).
פונקציות הבנאי וההרס עשויות להיות מוטבעות גם הן, כברירת מחדל, אם הן מוגדרות במחלקה שלהן, או באופן מפורש.

Parameterized Constructors page 307

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. For example, here is a simple class that includes a parameterized constructor:

בנאים עם פרמטרים
אפשר להעביר טיעונים לבנאים. בדרך כלל, ארגומנטים אלה עוזרים לאתחל אובייקט כאשר הוא נוצר. כדי ליצור בנאי בעל פרמטרים, פשוט הוסף לו פרמטרים כפי שהיית עושה לכל פונקציה אחרת. כאשר אתה מגדיר את גוף הבנאי, השתמש בפרמטרים כדי לאתחל את האובייקט. לדוגמה, הנה מחלקה פשוטה הכוללת בנאי בעל פרמטרים:

```
class myclass {
   int a, b;
```

```
  public:
    myclass(int i, int j) {a=i; b=j;}
    void show() {cout << a << " " << b;}
};
int main() {
    myclass ob(3, 5);
ob.show();
return 0; }
```

The program illustrates the most common way to specify arguments when you declare an object that uses a parameterized constructor. Specifically, this statement

התוכנית ממחישה את הדרך הנפוצה ביותר לציין ארגומנטים כאשר אתה מכריז על אובייקט שמשתמש בבנאי בעל פרמטרים. ספציפית, האמירה הזו

```
myclass ob(3, 4);
```

causes an object called ob to be created and passes the arguments 3 and 4 to the i and j parameters of myclass( ). You may also pass arguments using this type of declaration statement:

```
  myclass ob = myclass(3, 4);
```

However, the first method is the one generally used, and this is the approach taken by most of the examples in this book. Actually, there is a small technical difference between the two types of declarations that relates to copy constructors. (Copy constructors are discussed in Chapter 14.)

גורם ליצירת אובייקט בשם ob ומעביר את הארגומנטים 3 ו-4 לפרמטרים i ו-j של myclass( ). אתה יכול גם להעביר טיעונים באמצעות סוג זה של הצהרת הצהרה:

עם זאת, השיטה הראשונה היא השיטה הנהוגה בדרך כלל, וזו הגישה של רוב הדוגמאות בספר זה. למעשה, יש הבדל טכני קטן בין שני סוגי ההצהרות המתייחסים לבנאי העתקה. (בוני העתקה נדונים בפרק 14.)

Here is another example that uses a parameterized constructor. It creates a class that stores information about library books.

הנה דוגמה נוספת שמשתמשת בקונסטרוקטור בעל פרמטרים. זה יוצר מחלקה המאחסנת מידע על ספרי ספרייה.

```
#include <cstring>
 const int IN = 1;
```

```
  const int CHECKED_OUT = 0;
  class book {
    char author[40];
    char title[40];
    int status;
  public:
    book(char *n, char *t, int s);
    int get_status() {return status;}
    void set_status(int s) {status = s;}
    void show();
};
  book::book(char *n, char *t, int s)
  {
    strcpy(author, n);
 strcpy(title, t);
status = s; }

  void book::show()
     {
       cout << title << " by " << author;
       cout << " is ";
       if(status==IN) cout << "in.\n";
       else cout << "out.\n";
}
int main() {
       book b1("Twain", "Tom Sawyer", IN);
       book b2("Melville", "Moby Dick", CHECKED_OUT);
       b1.show();
       b2.show();
return 0; }
```

Parameterized constructors are very useful because they allow you to avoid having to make an additional function call simply to initialize one or more variables in an object. Each function call you can avoid makes your program more efficient. Also, notice that the short get_status( ) and set_status( ) functions are defined in line, within the book class. This is a common practice when writing C++ programs.

בנאים עם פרמטרים שימושיים מאוד מכיוון שהם מאפשרים לך להימנע מהצורך לבצע קריאת פונקציה נוספת פשוט כדי לאתחל משתנה אחד או יותר באובייקט. כל קריאת פונקציה שתוכל להימנע ממנה הופכת את התוכנית שלך ליעילה יותר. כמו כן, שימו לב שהפונקציות הקצרות get_status( ) ו-set_status( ) מוגדרות בשורה, בתוך מחלקת הספר. זהו נוהג נפוץ בעת כתיבת תוכניות C++.

Constructors with One Parameter: A Special Case

If a constructor only has one parameter, there is a third way to pass an initial value to that constructor. For example, consider the following short program.

קונסטרוקטים עם פרמטר אחד: מקרה מיוחד

אם לבנאי יש רק פרמטר אחד, יש דרך שלישית להעביר ערך התחלתי לאותו בנאי. לדוגמה, שקול את התוכנית הקצרה הבאה.

```
class X {
        int a;
public:
X(int j) { a = j; }
        int geta() { return a; }
      };
int main() {
        X ob = 99; // passes 99 to j
        cout << ob.geta(); // outputs 99
return 0; }
```

Here, the constructor for X takes one parameter. Pay special attention to how ob
is declared in main( ). In this form of initialization, 99 is automatically passed to the j parameter in the X( ) constructor.
That is, the declaration statement is handled by the compiler as if it were written like this:

ob לוקח פרמטר אחד. שימו לב במיוחד איך X כאן, הבנאי עבור j בצורה זו של אתחול, 99 מועבר אוטומטית לפרמטר .main( )-מוצהר ב בבנאי X( ).

כלומר, הצהרת ההצהרה מטופלת על ידי המהדר כאילו היא נכתבה כך:

```
X ob = X(99);
```

In general, any time you have a constructor that requires only one argument, you can use either ob(i) or ob = i to initialize an object. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.
Remember that the alternative shown here applies only to constructors that have exactly one parameter.

באופן כללי, בכל פעם שיש לך בנאי שדורש ארגומנט אחד בלבד, אתה יכול להשתמש ב-ob(i) או ob = i כדי לאתחל אובייקט. הסיבה לכך היא שבכל פעם שאתה יוצר בנאי שלוקח ארגומנט אחד, אתה גם יוצר באופן מרומז המרה מסוג הארגומנט לסוג המחלקה.

זכור שהחלופה המוצגת כאן חלה רק על בנאים שיש להם פרמטר אחד בדיוק.

Static Class Members

Both function and data members of a class can be made **static**. This section explains the consequences of each.

Static Data Members

When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created.

When you declare a **static** data member within a class, you are *not* defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated. (Remember, a class declaration is simply a logical construct that does not have physical reality.)
To understand the usage and effect of a **static** data member, consider this program:

חברי כיתה סטטיים
ניתן להפוך גם חברי פונקציה וגם נתונים של מחלקה לסטטיים. סעיף זה מסביר את ההשלכות של כל אחד מהם.
חברי נתונים סטטיים
כאשר אתה מקדים להצהרה של משתנה איבר עם סטטי, אתה אומר למהדר שרק עותק אחד של המשתנה הזה קיים ושכל האובייקטים של המחלקה יחלקו את המשתנה הזה. שלא כמו חברי נתונים רגילים, עותקים בודדים של משתנה איבר סטטי לא נוצרים עבור כל אובייקט. לא משנה כמה אובייקטים של מחלקה נוצרים, רק עותק אחד של חבר נתונים סטטי קיים. לפיכך, כל האובייקטים של אותה מחלקה משתמשים באותו משתנה. כל המשתנים הסטטיים מאותחלים לאפס לפני יצירת האובייקט הראשון.
כאשר אתה מצהיר על חבר נתונים סטטי בתוך מחלקה, אתה לא מגדיר

אותו. (כלומר, אינך מקצה לו אחסון.) במקום זאת, עליך לספק הגדרה גלובלית עבורו במקום אחר, מחוץ לכיתה. זה נעשה על ידי הצהרה מחדש של המשתנה הסטטי באמצעות אופרטור רזולוציית ההיקף כדי לזהות את המחלקה אליה הוא שייך. זה גורם להקצאת אחסון עבור המשתנה. (זכור, הצהרת מחלקה היא פשוט מבנה לוגי שאין לו מציאות פיזית).
כדי להבין את השימוש וההשפעה של חבר נתונים סטטיים, שקול את התוכנית הזו:

```
#include <iostream>
  using namespace std;
  class shared {
    static int a;
    int b;
  public:
    void set(int i, int j) {a=i; b=j;}
    void show();
};
int shared::a; // define a
  void shared::show()
  {
    cout << "This is static a: " << a;
    cout << "\nThis is non-static b: " << b;
    cout << "\n";
}


int main() {
shared x, y;
    x.set(1, 1); // set a to 1
    x.show();
    y.set(2, 2); // change a to 2
    y.show();
    x.show(); /* Here, a has been changed for both x and y
                because a is shared by both objects. */
return 0; }

other case:
int main() {
    // initialize a before creating any objects
    shared::a = 99; // static
    cout << "This is initial value of a: " << shared::a;
cout << "\n";
```

```
shared x;
    cout << "This is x.a: " << x.a;
return 0; }
```

Notice that the integer a is declared both inside shared and outside of it. As mentioned earlier, this is necessary because the declaration of a inside shared does not allocate storage.
שימו לב שהמספר השלם a מוצהר גם בתוך משותף וגם מחוצה לו. כפי שצוין קודם לכן, זה הכרחי מכיוון שההכרזה על משותף פנימי לא מקצה אחסון.
A static member variable exists before any object of its class is created. For example, in the following short program, a is both public and static. Thus it may be directly accessed in main( ). Further, since a exists before an object of shared is created, a can be given a value at any time. As this program illustrates, the value of a is unchanged by the creation of object x. For this reason, both output statements display the same value: 99.
משתנה איבר סטטי קיים לפני יצירת אובייקט כלשהו מהמחלקה שלו. לדוגמה, בתוכנית הקצרה הבאה, a הוא גם ציבורי וגם סטטי. כך ניתן לגשת ישירות אליו ב-main( ). יתרה מכך, מכיוון ש-a קיים לפני יצירת אובייקט משותף, ניתן לתת ערך בכל עת. כפי שתוכנית זו ממחישה, הערך של a אינו משתנה על ידי יצירת האובייקט x. מסיבה זו, שתי הצהרות הפלט מציגות את אותו ערך: 99.

Notice how a is referred to through the use of the class name and the scope resolution operator. In general, to refer to a static member independently of an object,
you must qualify it by using the name of the class of which it is a member.

One use of a static member variable is to provide access control to some shared resource used by all objects of a class. For example, you might create several objects, each of which needs to write to a specific disk file. Clearly, however, only one object can be allowed to write to the file at a time. In this case, you will want to declare a static variable that indicates when the file is in use and when it is free. Each object then interrogates this variable before writing to the file. The following program shows how you might use a static variable of this type to control access to a scarce resource:
שים לב כיצד מתייחסים ל-a באמצעות שימוש בשם המחלקה ובאופרטור רזולוציית ההיקף. באופן כללי, כדי להתייחס לאיבר סטטי ללא תלות באובייקט, עליך להכשיר אותו באמצעות שם המחלקה שבה הוא חבר.

שימוש אחד במשתנה חבר סטטי הוא לספק בקרת גישה למשאב משותף כלשהו המשמש את כל האובייקטים של מחלקה. לדוגמה, תוכל ליצור מספר אובייקטים, שכל אחד מהם צריך לכתוב לקובץ דיסק ספציפי. עם זאת, ברור שניתן לאפשר רק לאובייקט אחד לכתוב לקובץ בכל פעם. במקרה זה, תרצו להכריז על משתנה סטטי המציין מתי הקובץ נמצא בשימוש ומתי הוא פנוי. כל אובייקט חוקר את המשתנה הזה לפני הכתיבה לקובץ. התוכנית הבאה מראה כיצד תוכל להשתמש במשתנה סטטי מסוג זה כדי לשלוט בגישה למשאב נדיר:

```
#include <iostream>
  using namespace std;
  class cl {
    static int resource;
  public:
    int get_resource();
    void free_resource() {resource = 0;}
  };
  int cl::resource; // define resource
  int cl::get_resource()
  {
    if(resource) return 0; // resource already in use
    else {
resource = 1;
      return 1;  // resource allocated to this object
    }
}
int main() {
cl ob1, ob2;
    if(ob1.get_resource()) cout << "ob1 has resource\n";
    if(!ob2.get_resource()) cout << "ob2 denied resource\n";
    ob1.free_resource();  // let someone else use it
    if(ob2.get_resource())
      cout << "ob2 can now use resource\n";
return 0; }
```