國立清華大學
電機工程學系
108 學年度第一學期

EE-2310
計算機程式設計
Introduction to Programming

授課教師：黃錫瑜

2019年 Fall Semester

講義為課本作者及其出版社提供之投影片資料為基礎，
經授課教師修改之版本

Courtesy: These lecture notes are modified from a set of primitive PPT files provided by the textbook publisher.

國立清華大學 電機工程學系
Electrical Engineering Dept., National Tsing Hua University

Fall Semester, 2019

EE-2310 計算機程式設計 (Introduction to Programming)

教師 (Instructor): Prof. 黃錫瑜 (Shi-Yu Huang) (syhuang@ee.nthu.edu.tw)

Course Materials will be downloadable @【清華大學-數位學習系統】http://lms.nthu.edu.tw

Class Time: (M1M2R1R2) 8:00-9:50am

Classrooms：Lectures on Mondays @ Delta 217, Lectures on Thursdays @ Delta 215

Lab Sessions @ Delta 218, 219, and 220

一、課程說明 (General Description)

This course teaches the basics of computer programming in C++. In the first half, the syntax of C++ will be introduced. Students will learn the fundamental computer statements for mathematical and logical expressions, variable assignment, input/output, if-then-else control constructs, looping constructs, and various function calls. At the end of this stage, students should have been able to experience the joy of writing and executing a computer program. Then, in the second half, more advanced programming concepts will be followed, including Object-Oriented Programming (OOP) using Classes and Objects, data structure using Arrays, Strings, and Pointers, and finally fundamental Algorithms for Search and Sorting, and Recursive type of programming.

二、先修課程 (Prerequisites): None

三、主要課本 (Textbook):

- **Tony Gaddis, Judy Walters, and Godfrey Muganda, "*Starting Out with C++: Early Objects,*" 9th Edition, Addison-Wesley.**

四、助教 (Teaching Assistants): Delta Bldg. 923

楊竣宇(Jin-Yu Yang)	sclismeok@gmail.com	楊舜華 (Shun-Hua Yang)	a0987253982@gmail.com
蔡承霖 (Kris Tsai)	fd310021@gmail.com	林昂德 (Derek Lin)	bediligent300@gmail.com
許竹均 (Willian Hsu)	un8138@yahoo.com.tw		

五、教學內容大綱 (Outline of Topics) :

1. Introduction
2. Introduction to C++
3. Expressions
4. Make Decisions (Control Construct)
5. Looping
6. Functions
7. Classes and Objects
8. Arrays
9. Algorithm for Search and Sorting
10. Pointers
11. Basics of Object-Oriented Programming (OOP)
12. Strings
13. File IO Functions
14. Recursion

六、成績考核 (Grading Criteria): 上機作業(Homeworks) 40% 期中考(Midterm) 30% 期末考(Final) 30%

- There will be 8 Lab Sessions. One homework will be assigned at each Lab Session.
- Each Lab Session accounts for 5 points in the overall final score, attendance gets 2 points, demonstration of your homework assigned in the prior Lab Session gets 3 points.
- We will have an extra Lab Session for you to come to demo your last homework (HW#8) on Jan. 16, 2020.

七、考試資訊 (Exams):

期中考 (Midterm Exam.): 8:20-9:50am, @ Delta 215, Nov. 7 (Thursday), 2019.

期末考 (Final Exam.): 8:20-9:50am @ Delta 215, Jan. 9 (Thursday), 2020.

附表 (Appendix)：EE-2310 Class Schedule

Fall Semester, 2019

課堂上課 (Lectures): Monday @ Delta 217, Thursday @ Delta 215

上機實習課 (Lab Sessions): @ Delta 218, 219, and 220.

Lab Sessions @ Delta 218: Student ID No. smaller than **108000111**

Lab Sessions @ Delta 219: Student ID No. between **[108000111, 108061181]**

Lab Sessions @ Delta 220: Student ID No. larger than **108061181**

Week	Month	星期一 (Monday)	星期二 (Tuesday)	星期三 (Wednesday)	星期四 (Thursday)	星期五 (Friday)
1	Sept 2019	Sept. 9 (Lecture)	Sept. 10	Sept. 11	Sept. 12 (Lecture)	Sept. 13
2		Sept. 16 (Lecture)	Sept. 17	Sept. 18	Sept. 19 (Lab #1)	Sept. 20
3		Sept. 23 (Lecture)	Sept. 24	Sept. 25	Sept. 26 (Lecture)	Sept. 27
4	Oct 2019	Setp. 30 (Lecture)	Oct. 1	Oct. 2	Oct. 3 (Lab #2)	Oct 4
5		Oct. 7 (Lecture)	Oct. 8	Oct. 9	Oct. 10 (雙十 Holiday No Class)	Oct. 11
6		Oct. 14 (Lecture)	Oct. 15	Oct. 16	Oct. 17 (Lab #3)	Oct. 18
7		Oct. 21 (Lecture)	Oct 22	Oct. 23	Oct. 24 (Lab #3)	Oct. 25
8		Oct. 28 (Lecture)	Oct. 29	Oct. 30	Oct. 31 (Lab #4)	Nov. 1
9	Nov 2019	Nov. 4 (Lecture)	Nov. 5	Nov. 6	Nov. 7 期中考 8:20-9:50am	Nov. 8
10		Nov. 11 (教師出國開會 no class)	Nov. 12	Nov. 13	Nov. 14 (教師出國開會 no class)	Nov. 15
11		Nov. 18 (Lecture)	Nov. 19	Nov. 20	Nov. 21 (Lab #5)	Nov. 22
12		Nov. 25 (Lecture)	Nov. 26	Nov. 27	Nov. 28 (Lecture)	Nov. 29
13	Dec 2019	Dec. 2 (Lecture)	Dec. 3	Dec. 4	Dec. 5 (Lab #6)	Dec. 6
14		Dec. 9 (Lecture)	Dec. 10	Dec. 11	Dec. 12 (Lecture)	Dec. 13
15		Dec. 16 (Lecture)	Dec. 17	Dec. 18	Dec. 19 (Lab #7)	Dec. 20
16		Dec. 23 (Lecture)	Dec. 24	Dec. 25	Dec. 26 (Lecture)	Dec. 27
17		Dec. 30 (Lecture)	Dec. 31	Jan. 1	Jan. 2 (Lab #8)	Jan. 3
18	Jan 2020	Jan. 6 (期末週 no class)	Jan. 7	Jan. 8	Jan. 9 期末考 8:20-9:50am	Jan. 10
19					Jan. 16 (HW#8 驗收)	

```cpp
// Lab 1: First Program
#include <iostream>
#include <stdio.h>
#define PI 3.14159

using namespace std;

int main()
{
    // Print out a welcome message
    cout << "Hello world!" << endl;
    string  first_name, last_name;
    cout << "Please enter your <first_name> <last_name>: ";
    cin >> first_name >> last_name;
    cout << first_name << " " << last_name << endl;
    cout << "Welcome to the world of C++ ..." << endl;

    // Compute the area of a round shape of a given diameter
    float diameter, area;
    // float pi = 3.14159;
    cout << "Please enter the diameter of a round shape: ";
    cin >> diameter;
    area = PI * (diameter)*diameter;
    cout << "The area of a round shape with a diameter of " << diameter <<
" is " << area << endl;

    // Another way of printing message on the DISPLAY
    cout << "Let's print the message using C-function printf..." << endl;
    printf("The area of a round shape with a diameter of %f is %f\n",
diameter, area);

    return 0;
}

```

```
// Lab 2: If-then-Else Control
#include <iostream>
#include <stdio.h>
#include <cmath>
#include <iomanip>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;

    /***** The sizes of basic data types *****/
    printf("The size of SHORT, INT, and LONG INT are: %d, %d, %d BYTES\n",
           sizeof(short), sizeof(int), sizeof(long int));
    printf("The size of FLOAT, DOUBLE, and LONG DOUBLE are: %d, %d, %d BYTES\n",
           sizeof(float), sizeof(double), sizeof(long double));

    /***** Witness of Overflow *****/
    cout << endl;
    int  largest = pow(2, 15)-1;
    cout << "Assigning NUM = largest + 1" << endl;
    printf("Largest integer in 16 bits (in INT) is: %d\n", largest);
    int NUM = (largest + 1);
    cout << "Display NUM in SHORT INT: " << (short) NUM << endl; // Overflow
    cout <<  "Display NUM in INT: " << NUM << endl;

    cout << right << setw(30);
    cout << setprecision(8);
    cout << float(NUM) << endl;

    /***** Display a real number *****/
    cout << "Display NUM in FLOAT: " << float(NUM) << endl; // Type Casting
    printf("Display NUM in Fixed_Point notation: %10.5f\n", float(NUM));
    printf("Display NUM in Scientific notation using formatted .2e: %.2e\n", float(NUM));
    printf("Display NUM in Scientific notation using Static-Cast: %.2e\n", static_cast<double>(NUM));

    /***** If-then-else control construct *****/
    cout << endl << "Please enter two integers less than 100: ";
    int v1, v2;
    cin >> v1 >> v2;
    if(v1<1 || v2<1){
        cout << "You have entered an illegal integer smaller than 1";
    }
    else if (v1>100 || v2>100){
        cout << "You have entered illegal integer(s) larger than 100";
    }
    else {
        cout << "Thank you! You have entered legal integers, ";
        cout << v1 << " " << v2 << endl;
        // Do the integer division here
        int q, r;
        q = v1 / v2;
        r = v1 % v2;
        printf("Dividing %d by %d will give you (Quotient, Remainder) = (%d, %d)\n", v1, v2, q, r);
    }
    return 0;
}

```

```
// Lab3: Looping and GCD Finder
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

using namespace std;

int find_gcd(int, int); // function prototyping

int main()
{
    int  i;
    cout << "Hello world!" << endl;

    /***-- Creating random numbers in an array --***/
    int  A[5] = {102, 340, 153, 187, 425};
    for(i=0; i<5; i++) printf("%d, ", A[i]); cout << endl;

    srand(time(0));
    for(i=0; i<5; i++){ A[i] = rand()%100 + 10; }
    cout << "Randomly generated numbers: ";
    for(i=0; i<5; i++){ printf("%d, ", A[i]); }
    cout << endl << endl;

    /***-- Print the binary representation of a number ---***/
    char ch = 'a';
    printf("The binary representation for %c is: ", ch);
    for(i=7; i>=0; i--){
        cout << ((ch >> i) & 1);
    }
    cout << endl;

    /***-- Switch on Enumeration Type of Data --***/
    enum WeekDay {Mon, Tue, Wed, Thu, Fri, Sat, Sun}; // Enumerated Constant Variables
    WeekDay day;
    day = Mon;
    switch(day){
            case Mon: cout << "Monday" << endl; break;
            case Tue: cout << "Tuesday" << endl; break;
            case Wed: cout << "Wednesday" << endl; break;
            case Thu: cout << "Thursday" << endl; break;
            case Fri: cout << "Friday" << endl; break;
            case Sat: cout << "Saturday" << endl; break;
            case Sun: cout << "Sunday" << endl; break;
            default: cout << "Out of range WeekDay value!" << endl; break;
    }
    cout << endl;

    /***-- Produce GCD for 2 Integers --***/
    int num1, num2, gcd;
    num1 = 2*3*3*7; num2 = 3*5*7;
    printf("Two random numbers are (%d, %d)\n", num1, num2);
    gcd = find_gcd(num1, num2); // call the sub-routine here
    printf("The GCD of (%d, %d) is: %d\n", num1, num2, gcd);
}
/*------- A subroutine for calculating the GCD of two integer numbers ---------*/
int find_gcd(int n1, int n2)
{
    int x1=n1, x2=n2;
    int gcd;
    while(1){
        if(x1==0) { gcd = x2; break; }
        if(x2==0) { gcd = x1; break; }
```

```
          /*--- Three conditions to explore ---*/
          if(x1==x2) { gcd = x1; break; }
          else if(x1>x2) x1 = x1%x2; // n1 is the larger number
          else  x2 = x2%x1; // n2 is the larger number
      }

      return gcd;
  }

```

```cpp
// Lab4: Reading a line and Call-by-Reference (Swapping)
#include <iostream>
#include <string>
#include <fstream>
#include <stdlib.h>
#include <string.h>

using namespace std;

void print_array(string, int*, int); // function prototyping
void c_swap(int*, int*); // function prototyping
void swap(int&, int&); // function prototyping

int main(int argcount, char* argv[])
{
    printf("Total number of command line arguments: %d\n", argcount);
    printf("Argv[0] is %s\n", argv[0]);
    printf("Argv[1] is %s\n", argv[1]);

    // Reading two integers from a file called lab.data
    // ifstream InFile ("lab4.txt");
    ifstream InFile (argv[1]);
    if (!InFile) exit(-1);

    // Process one line at a time
    int     i=0;
    string line = "111";
    char    c_line[1000];
    int     A[1000]; // for storing the input data
    while(getline(InFile, line)){
        /* convert a C++ line to a C string */
        strcpy(c_line, line.c_str());
        sscanf(c_line, "%d", &A[i]);
        printf("A[%d] is: %d\n", i, A[i]);
        i++;
    }
    InFile.close();
    int size =i; // record the total number of data
    printf("The total number of data is: %d\n", size);
    print_array("Original Data Stream:", A, size);

    // Swap two A[0] and A[1]
    c_swap(&A[0], &A[1]);
    print_array("After swapping the first two data via call-by-address:", A, size);

    swap(A[0], A[1]);
    print_array("After swapping again the first two data via call-by-reference:", A, size);

    // How to use a 2-dimensional array
    int B[3][3]={{1,2,3},{4,5,6},{7,8,9}};
    cout << "*****************************************" << endl;
    cout << "Display the contents of a 2-D array: " << endl;
    for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
                cout << B[i][j] << "  ";
        }
        cout << endl;
    }

    return 0;
}
void print_array(string message, int *A, int n)
{
    int i;
    cout << "****************************************" << endl;
    cout << message << endl;
    for(i=0; i<n; i++){
        printf("%4d  ", A[i]);
    }
    cout << endl;
}
void c_swap(int *ptr_x1, int *ptr_x2)
{
    int temp;
    temp = *(ptr_x1);
    *(ptr_x1) = *(ptr_x2);
    *(ptr_x2) = temp;
}
void swap(int& x1, int& x2)
{
    int temp;
    temp = x1;
    x1 = x2;
    x2 = temp;
}

```

```cpp
// Lab5: Class and Object
#include <iostream>
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>

using namespace std;
void show_string(HANDLE&, int, int, string);
void clear_screen(HANDLE&, int, int);
void sleep(int);

class complex {
private:
    double  real;
    double  imaginary;
public:
    complex(double re = 0.0, double im =0.0){
        real = re;
        imaginary = im;
    }
    void   set_a_complex(double re, double im){
        real = re;
        imaginary = im;
    }
    double  get_real(){ return(real);}
    double  get_imaginary(){ return(imaginary);}
    complex  operator+(complex x){
        complex sum;
        double  final_real = this->get_real() + x.get_real();
        double  final_imaginary = this->get_imaginary() + x.get_imaginary();
        sum.set_a_complex(final_real, final_imaginary);
        return(sum);
    }
    int operator==(complex& b){
        bool if_eq = (this->get_real()==b.get_real() && this->get_imaginary()==b.get_imaginary());
        return(if_eq);
    }
};

// overload << as a regular function
// Function Prototype + Function Definition
ostream& operator<<(ostream& os, complex &x){
    os << "complex number: " << x.get_real() << " + j " << x.get_imaginary() << endl;
    return(os);
};


int main()
{
    complex    a(1, 2), b(10, 20) ;
    a.set_a_complex(1, 2);
    b.set_a_complex(1, 2);
    cout << "Checking if a is equal to b: " << (a==b) << endl;
    complex sum = a + b;
    cout << a << b << sum;

    /*----------- Display character in a particular position on the screen --------*/
    HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);

    clear_screen(screen, 20, 100);
    for(int k=0; k<11; k++){
          show_string(screen, k, k, "x");
    }
    sleep(5);
    for(int k=0; k<11; k++){
         show_string(screen, k, 10-k, "x");
    }
    show_string(screen, 12, 0, "****************");
}

void clear_screen(HANDLE& screen, int row_count, int col_count)
{
    string  s = "";
    for(int j=0; j<col_count; j++){
        s += " ";
    }
    for(int i=0; i<row_count; i++){
        show_string(screen, i, 0, s);
    }
}
void show_string(HANDLE& screen, int row, int col, string s)
{
    COORD   position;
    position.Y = row;
    position.X = col;
    SetConsoleCursorPosition(screen, position);
    cout << s ;
}
void sleep(int second){
    for(int i=0;i<second*100000000; i++){
        i=i;
    }
}
```

```cpp
// Lab6.1: Binary Search
#include <iostream>
#include <stdio.h>

using namespace std;
void show_array(int *A, int size);

int main()
{
    cout << "Hello world!" << endl;

    int A[5]={5, 20, 33, 45, 67};
    int target = 33;
    int left, right, pivot;
    left = 0, right = 4;
    cout << "Original Array\n" ;
    show_array(A, 5);
    while(1)
    {
        pivot = (left+right)/2;
        printf("(Left, Right, Pivot) = (%d, %d - %d)\n", left, right, pivot);
        if(left>right) { cout << "Target not found!" << endl; return(-1); }
        if(A[pivot]==target) {
                cout << "Target Found! " << A[pivot] << " at array index " << pivot <<  endl; return(pivot);
        }
        else if (target < A[pivot]) {
                right = pivot - 1;
        }
        else { //  (target < A[pivot])
                left = pivot + 1;
        }
    }
}
void show_array(int *A, int size)
{
    int i;
    for(i=0; i<size; i++){
        cout << A[i] << " ";
    }
    cout << endl;
}
```

```cpp
// Lab6.2: Selection Sort
#include <iostream>
#include <stdio.h>

using namespace std;
void show_array(int A[], int size);

int main()
{
    cout << "Hello world!" << endl;

    int A[6]={110, 5, 20, 9, 55, 88};
    int i, j, size=6;
    int temp, smallest;
    cout << "Original Array:\n";
    show_array(A, 6);
    for(i=0; i<size; i++)
    {
        smallest = i;
        for(j=i; j<size; j++){
            if(A[j]<A[smallest]) smallest = j;
        }
        printf("The smallest one is: %d\n", smallest);
        if(smallest != i){
            temp = A[i];
            A[i]=A[smallest];
            A[smallest] = temp;
        }
    }
    cout << "Sorted Array:\n";
    show_array(A, 6);

}
void show_array(int A[], int size)
{
    for(int i=0; i<size; i++){
        cout << A[i] << " " ;
    }
    cout << endl;
}

```

```
// Lab6.3: Bubble Sort
#include <iostream>
#include <stdio.h>

using namespace std;
void show_array(int A[], int size);
void swap(int &a, int &b);

int main()
{
    cout << "Hello world!" << endl;

    int A[6]={10, 9, 8, 7, 6, 5};
    int i, size=6;
    int position_to_fix;
    cout << "Original Array:\n";
    show_array(A, size);
    for(position_to_fix=(size-1); position_to_fix>0; position_to_fix--)
    {
        for(i=0; i<position_to_fix; i++)
        {
            if(A[i] > A[i+1]){
                swap(A[i], A[i+1]);
                // printf("Swapping at %d (%d, %d)\n", i, A[i], A[i+1]);
            }
        }
    }
    cout << "Sorted Array:\n";
    show_array(A, size);
}
void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
void show_array(int A[], int size)
{
    for(int i=0; i<size; i++){
        cout << A[i] << " " ;
    }
    cout << endl;
}

```

```
// Lab6.4: Sorting using <Vector> in STL
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <algorithm>

using namespace std;
void show_array(vector<int> A);
int myfunction(int& x1, int& x2)
{
    return(x1 < x2);
}

  main()
{
    cout << "Hello world!" << endl;

    vector<int> A;
    int size = 10;

    for(int i=0; i<size; i++)
        A.push_back(rand()%100);
    cout << "Original Array:\n";
    show_array(A);
    sort(A.begin(), A.end(), myfunction);
    cout << "Sorted Array:\n";
    show_array(A);

}
void show_array(vector<int> A)
{
    for(int i=0; i<(int) A.size(); i++)
    {
        cout << A[i] << " " ;
    }
    cout << endl;
}

```

```cpp
// Lab7: File IO and Parsing Article
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <string>
#include <string.h>
#include <vector>
#include <algorithm>

using namespace std;

#define VERBOSE 0

void process_a_line(string cpp_line, vector<string>& A);
void process_a_word(string cpp_word, vector<string>& A);
void print_word_array(vector<string>&, int top);

/*****************************************/
/* Comparison Function for sorting Alg.  */
/*****************************************/
int myfunction(string p1, string p2){
    int r = (p1 < p2);
    // int r=(strcmp(p1.c_str(), p2.c_str())<0) ? 1 : 0;
    return(r);
}

ofstream out_fp("dict.out", ios::out);

int main(int argc, char **argv)
{

if(argc<2){
    cout << "Too few arguments" << endl; return(-1);
    printf("Usage: %s <filename>\n", argv[0]);
}
 ifstream in_fp(argv[1], ios::in); // creating input file handle
 if(! in_fp) { cout << "Input file " << argv[1] << " is not valid\n" <<
endl;  return(0); }

 string cpp_line;
 int    line_count=0;


 // Reading all words in a file into a vector of strings
 vector<string>  A;
 string  p;

 getline(in_fp, cpp_line);

 while(!cpp_line.empty()){
    if(VERBOSE>0){
        printf("The %d-th line:\n", line_count);
        cout << cpp_line << endl;
    }
    process_a_line(cpp_line, A);
    line_count++;
    getline(in_fp, cpp_line);
 }
 out_fp << "Total no. of lines: " << line_count << endl;
 char  buffer[1000];
 sprintf(buffer, "The total number of words in %s: %d\n", argv[1],
A.size());
 out_fp << buffer;

 /*--- Sort the vector of words-and-counts ---*/
 sort(A.begin(), A.end(), myfunction);
 print_word_array(A, 1000);
}
/*----------- process a line -------------*/
void print_word_array(vector<string>& A, int top)
{
    int i;
    for(i=0; i<top && i<(int) A.size(); i++){
        printf("The %d-th word is (%s)\n", i+1, A[i].c_str());
        out_fp << (i+1) << " -th word is " << A[i].c_str() << endl;
    }
}
/*----------- process a line -------------*/
void process_a_line(string cpp_line, vector<string>& A)
{
    char c_line[1000000], *word;
    string cpp_word;

    if(cpp_line == "\n") { cout << "An empty line!\n"; }
    strcpy(c_line, cpp_line.c_str());
        word = strtok(c_line, "\"-,:;.() ");
    while(word != 0){
        /************************************/
        /*         Process a word here      */
        /************************************/
        cpp_word = word;
        process_a_word(cpp_word, A);
        word = strtok(NULL, "\"-,:;.() ");
    }
}
/*----------- process a line -------------*/
void process_a_word(string cpp_word, vector<string>& A)
{
    bool if_exist = false;
    string p;
    int i;

    /*--- check if cpp_word exists in the vector ---*/
    for(i=0; i<(int) A.size(); i++){
        if(cpp_word==A[i]){
            if_exist = true;
            break;
        }
    }
    if(if_exist==false){ // add a new word
        A.push_back(cpp_word);
    }
}


```

```
// Lab7 (Version 2): Using <Map> in STL for Parsing an Articles
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <map>
#include <string>
#include <string.h>
#include <vector>
#include <algorithm>

#define VERBOSE 0

using namespace std;
class name_freq_pair{
    public:
        string  name;
        int     freq;
};
void process_a_line(string, map<string, int>&);
void print_word_array(vector<name_freq_pair>&, int);

int myfunction(name_freq_pair & p1, name_freq_pair & p2){
    return(p1.freq > p2.freq);
}

int main(int argc, char **argv)
{

if(argc<2){
    cout << "Too few arguments" << endl; return(-1);
    printf("Usage: %s <filename>\n", argv[0]);
}
 ifstream in_fp(argv[1], ios::in);
 if(! in_fp) { cout << "Input file " << argv[1] << " is not valid\n" <<
endl;  return(0); }

 printf("%s %s\n", argv[0], argv[1]);
 // create a map
 map<string,int> book;
 map<string,int>::iterator it;

 string cpp_line;
 int    line_count=0;

 // Building dictionary
 getline(in_fp, cpp_line);
 if(VERBOSE>0){ cout << cpp_line << endl; }
 while(!cpp_line.empty()){
    process_a_line(cpp_line, book);
    line_count++;
    getline(in_fp, cpp_line);
 }
 cout << "Total no. of lines: " << line_count << endl;

 /*--- convert the map into a vector ---*/
 vector<name_freq_pair>  A;
 name_freq_pair  p;
 for(it=book.begin(); it!=book.end(); it++){
    // word and its count
    if(VERBOSE>0){  cout << it->first << "  " << it->second << endl; }
    p.name = it->first;
    p.freq = it->second;
    A.push_back(p);
 }
 printf("The total number of words in %s: %d\n", argv[1],  A.size());

 /*--- Sort the vector of words-and-counts ---*/
 sort(A.begin(), A.end(), myfunction);
 print_word_array(A, 10);
}
/*----------- process a line -------------*/
void print_word_array(vector<name_freq_pair>& A, int top)
{
    int i;
    for(i=0; i<top && i<(int) A.size(); i++){
        printf("The %d-th frequent word is (%s) with no. of appearances 
(%d)\n", i, A[i].name.c_str(), A[i].freq);
    }
}
/*----------- process a line -------------*/
void process_a_line(string cpp_line, map<string, int>& book)
{

    char c_line[1000000], *word;
    string cpp_word;
    map<string,int>::iterator it;

    strcpy(c_line, cpp_line.c_str());
    word = strtok(c_line, "\"-,:;.() ");
    while(word != 0){
        /************************************/
        /*         Process a word here      */
        /************************************/
        cpp_word = word;
        it = book.find(cpp_word);
        if(it != book.end()){
            // An existing word
            it->second = (it->second)+1;
        }
        else {// A new word
            book[cpp_word]=1;
        }
        word = strtok(NULL, "\"-,:;.() ");
    }
}


```

```
// Lab8: Recursive Algorithms - Factorial, Combinatorial,
//                                Binary Search, Quick Sort
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <string.h>

using   namespace std;
int     factorial(int x);
int     comb(int m, int n);
int     recursive_binary_search(int *a, int target, int left, int right);
void    show_array(const char message[], int *A, int size);
void    quick_sort(int *B, int left, int right);\
void    swap(int *B, int i, int j);


int main()
{
    int     i, n=10;

    /*--- compute factorial recursively ---*/
    for(i=1; i<=n; i++){
        int f = factorial(i);
        printf("The factorial of (%d) is (%d)\n", i, f);
    }
    /*--- compute combinatorial recursively ---*/
    /* C(m, n) = C(m, n-1) + C(m-1, n-1) */
    cout << endl;
    int m;
    for(m=1; m<=n; m++){
        printf("The Combinatorial Number of selecting %d out of %d is (%d)\n",
               m, n, comb(m, n));
    }
    /*--- recursive binary search ---*/
    int     A[10]={1, 3, 6, 10, 22, 33, 41, 45, 55, 92};
    int     x = 33; // element to be sought from the array
    printf("\nSearch for an element (%d) from an array of (%d) elements ...\n", x, n);
    show_array("Array is: ", A, 10);
    int found = recursive_binary_search(A, x, 0, 9);
    printf("Search Result: Found = (%d)\n", found);

    /*--- recursive quick sort ---*/
    int     B[15]={7, 6, 3, 2, 9, 8, 1, 4, 5, 10, 8, 9, 3, 5, 6};
    show_array("Original Array is:      ", B, 15);
    quick_sort(B, 0, 14);
    show_array("Array after Sorting is: ", B, 15);
}

/***-------------- Binary Search -----------***/
int recursive_binary_search(int *a, int target, int left, int right)
{
    int found;

    printf("Search in [%d, %d] for element (%d)\n", left, right, target);

    if(left > right) return(-1); // base condition, return

    int middle = (left + right) / 2;
    if(target==a[middle]) return(middle); // found the element, return the index

    if(target < a[middle])
        found = recursive_binary_search(a, target, left, middle-1);
    else
```

```
        found = recursive_binary_search(a, target, middle+1, right);
    return(found);
}

/***-------------- combinatorial -----------***/
int comb(int m, int n)
{
    if(m<=0 || n<=0 || m>n){
        cout << "Something's wrong" << endl;
        exit(-1);
    }
    if(m==n)     return(1);
    if(m==1)     return(n);
    if(n==1)     return(1);

    return(comb(m, n-1)+comb(m-1, n-1));
}
/***-------------- Factorial -----------***/
int factorial(int n)
{
    /*--- compute n-factorial ---*/
    if(n==0)     return(1);
    if(n==1)     return(1);
    else     return(factorial(n-1)*n);
}
/***-------------- Quick Sort -----------***/
void quick_sort(int *B, int left, int right)
{
    if(left >= right) return; // done
    printf("(left, right) = (%d, %d)\n", left, right);
    int i = left+1;
    int j = right;
    while(1){
        // find next larger element than the middle element in LEFT region
        for(; B[i]<B[left]; i++);
        // find next smaller element than the middle element in RIGHT region
        for(; B[j]>B[left]; j--);
        if(i<j){
            swap(B, i, j);
            show_array("After swapping ", B, 10);
        }
        else{
            swap(B, left, j);
            show_array("        After swapping ", B, 10);
            break;
        }
    }
    // recursive calls here
    quick_sort(B, left, j-1);
    quick_sort(B, j+1, right);
}
/***-------------- Show an array -----------***/
void show_array(const char message[], int *A, int size)
{
    printf("%s", message);
    for(int k=0; k<size; k++){ cout << A[k] << " "; }
    cout << endl;
}
/***-------------- Swap two element in an array -----------***/
void swap(int *B, int i, int j)
{
    printf("Swapping (index, value) = (%d, %d) <-> (%d, %d)\n", i, B[i],
j, B[j]);
    int temp=B[i]; B[i]=B[j]; B[j]=temp;
}

```

Chapter 1: Introduction to Computers and Programming

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

Addison-Wesley is an imprint of PEARSON

Topics

1.1 Why Program?
1.2 Computer Systems: Hardware and Software
1.3 Programs and Programming Languages
1.4 What Is a Program Made of?
1.5 Input, Processing, and Output
1.6 The Programming Process

1.1 Why Program?

Computer – programmable machine designed to follow instructions

Program – A sequence of statements, if followed, accomplish the computation of a specific task

Programmer – person who writes instructions (programs) to make computer perform a task

SO, without programmers, no programs; without programs, the computer cannot do anything

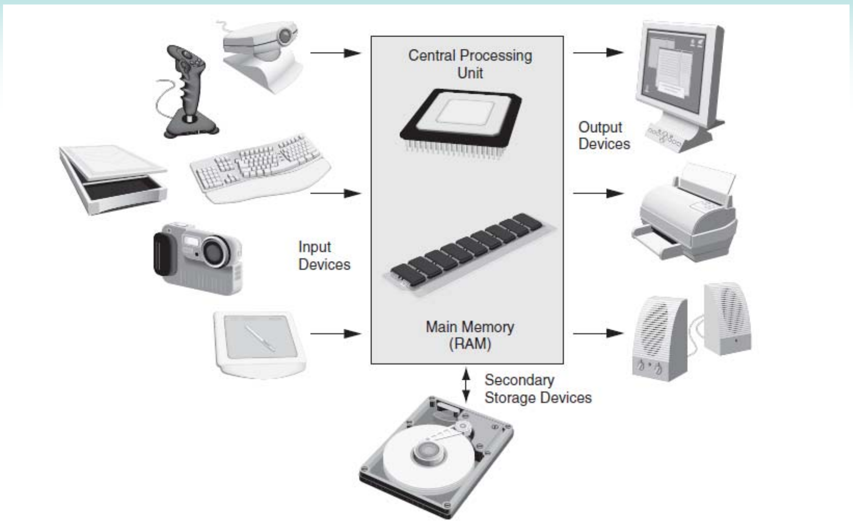
1.2 Computer Systems: Hardware and Software

Main Hardware Component Categories

1. Central Processing Unit (CPU)
2. Main memory (RAM)
3. Secondary storage devices
4. Input Devices
5. Output Devices

RAM: stands for Random Access Memory
-A piece of data in the memory of an arbitrary location can be accessed with a similar access time.

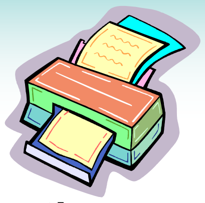
Main Hardware Component Categories



Input Devices

- **Used to send information <u>to the computer from outside</u>**
- **Many devices can provide input**
  - **keyboard, mouse, microphone, scanner, digital camera, disk drive, CD/DVD drive, USB flash drive**

Output Devices

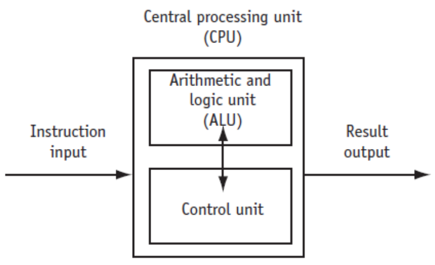


- Used to send information from the computer to the outside
- Many devices can be used for output
  - Computer screen, printer, speakers, disk drive, CD/DVD recorder, USB flash drive

Central Processing Unit (CPU)

Includes

- Control Unit
  - Retrieves and decodes program instructions
  - Coordinates computer operations
- Arithmetic & Logic Unit (ALU)
  - Performs mathematical operations

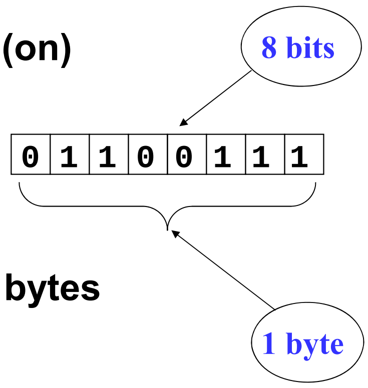

Central processing unit (CPU)
Arithmetic and logic unit (ALU)
Control unit
Instruction input
Result output

Main Memory

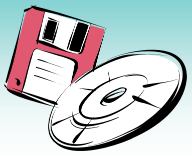
- Holds both program instructions and data (original data and results)
- Volatile – erased when program terminates or computer is turned off
- Often Random Access Memory (RAM)

Main Memory Organization

- Bit
  - Smallest piece of memory
  - Stands for binary digit
  - Has values 0 (off) or 1 (on)
- Byte
  - Is 8 consecutive bits
- Word
  - Usually 4 consecutive bytes
  - Has an address



Secondary Storage



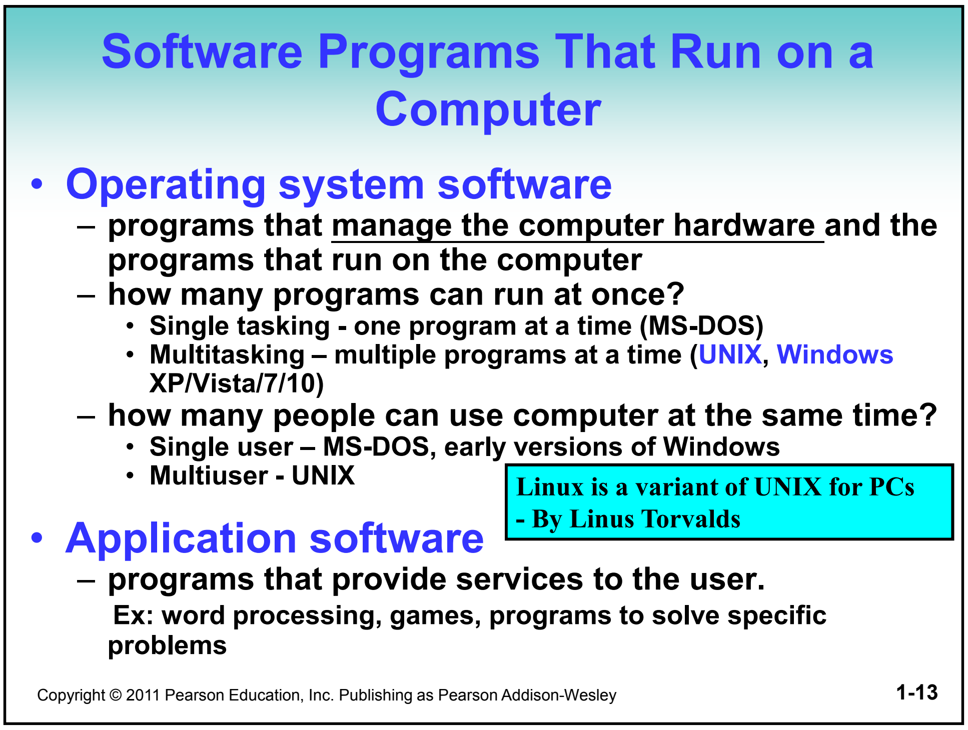
- Non-volatile - data retained when program is not running or computer is turned off
- Comes in a variety of media
  - magnetic: floppy or hard disk drive, internal or external
  - optical: CD or DVD drive
  - flash: USB flash drive

USB flash drive
Or SSD (Solid-State Disk)
is made of flash memory

The CPU's Role in Running an Instruction

4 Steps in Sequence:

- Fetch: get the next program instruction from main memory 抓取指令
- Decode: interpret the instruction and generate a signal 解碼指令
- Execute: route the signal to the appropriate component to perform an operation 執行指令
- Write Back: route the result back to the memory 儲存結果



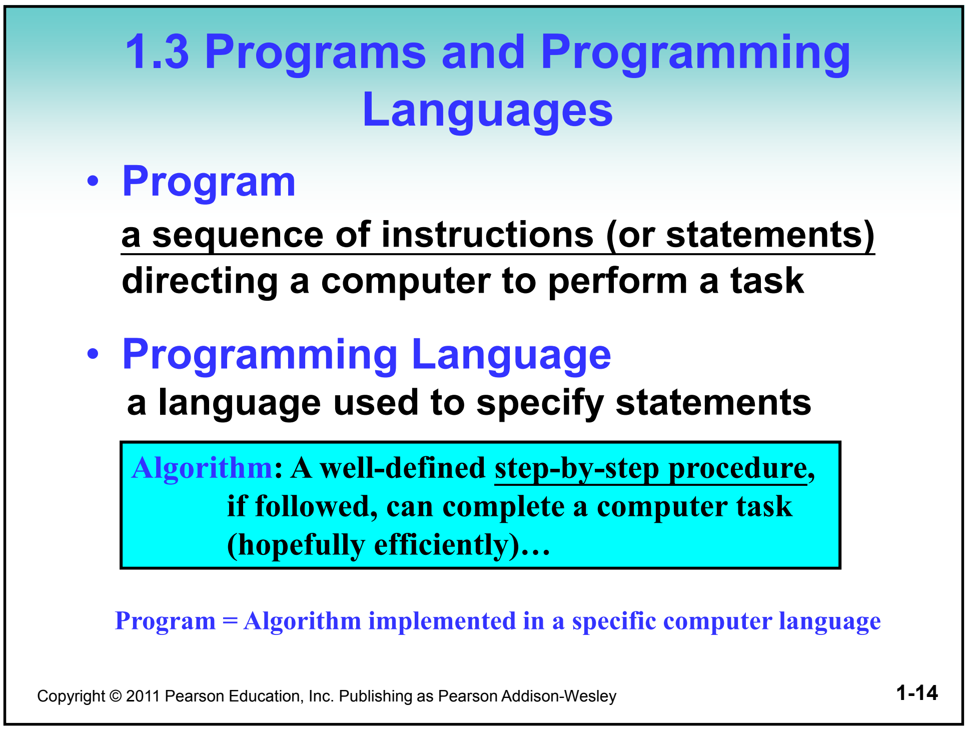
Software Programs That Run on a Computer

- Operating system software
  - programs that manage the computer hardware and the programs that run on the computer
  - how many programs can run at once?
    - Single tasking - one program at a time (MS-DOS)
    - Multitasking – multiple programs at a time (UNIX, Windows XP/Vista/7/10)
  - how many people can use computer at the same time?
    - Single user – MS-DOS, early versions of Windows
    - Multiuser - UNIX

Linux is a variant of UNIX for PCs - By Linus Torvalds

- Application software
  - programs that provide services to the user.

    Ex: word processing, games, programs to solve specific problems



1.3 Programs and Programming Languages

- Program

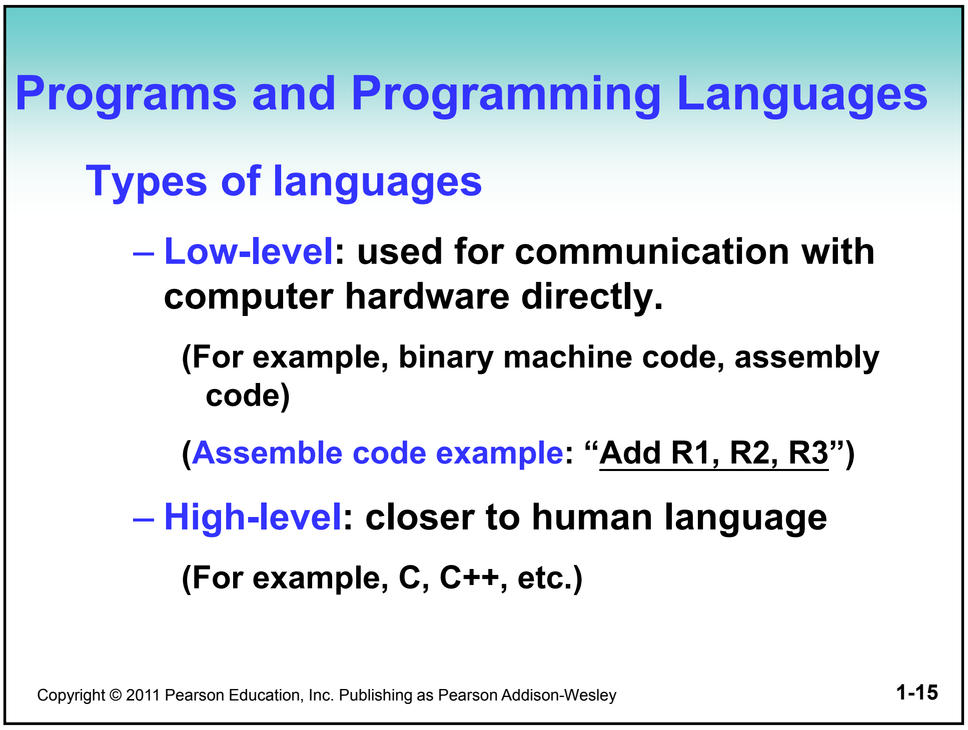
  a sequence of instructions (or statements) directing a computer to perform a task

- Programming Language

  a language used to specify statements

Algorithm: A well-defined step-by-step procedure, if followed, can complete a computer task (hopefully efficiently)…

Program = Algorithm implemented in a specific computer language



Programs and Programming Languages

Types of languages

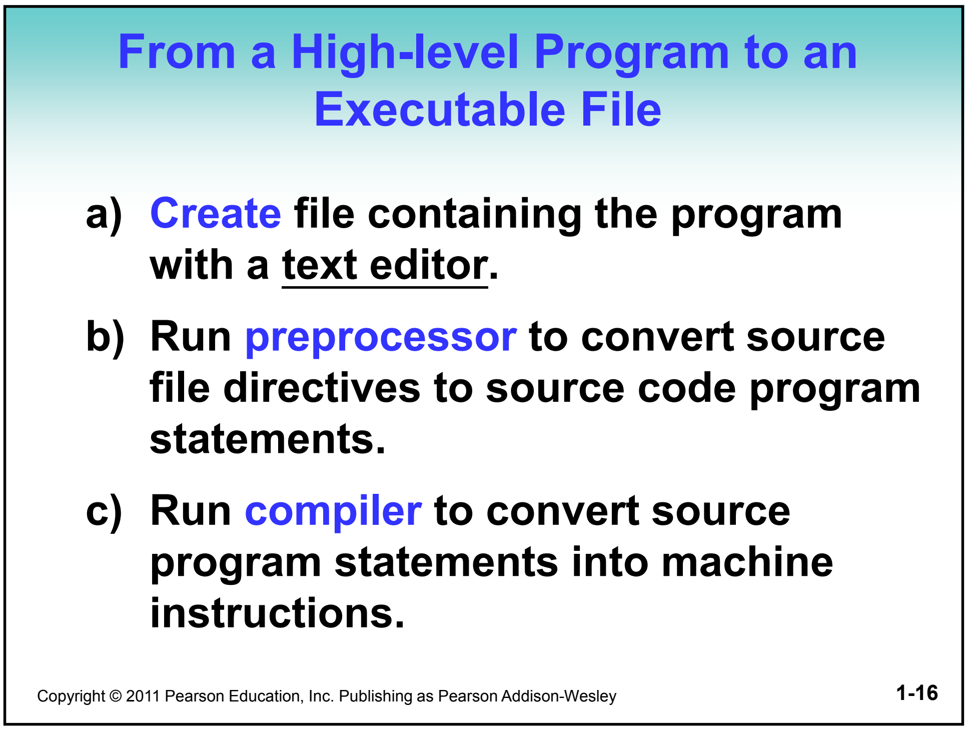
- Low-level: used for communication with computer hardware directly.

  (For example, binary machine code, assembly code)

  (Assemble code example: "Add R1, R2, R3")

- High-level: closer to human language

  (For example, C, C++, etc.)



From a High-level Program to an Executable File

a) Create file containing the program with a text editor.

b) Run preprocessor to convert source file directives to source code program statements.

c) Run compiler to convert source program statements into machine instructions.

From a High-level Program to an Executable File

d) Run linker to connect hardware-specific library code to machine instructions, producing an executable file.

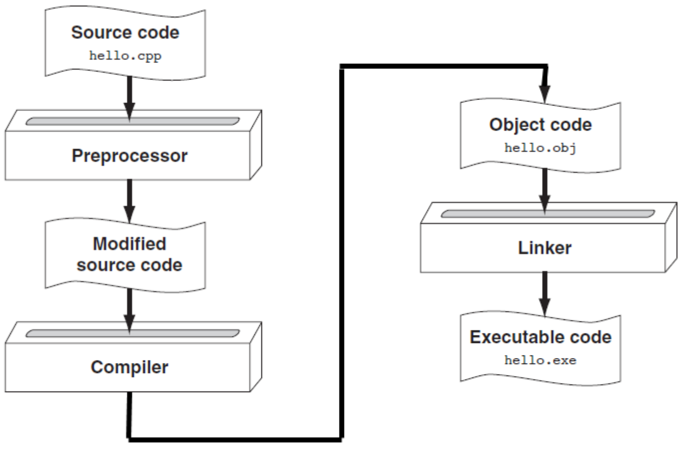
Steps b) through d) are often performed by a single command or button click.

Errors occurring at any step will prevent the execution of the following steps.

1-17

From a High-level Program to an Executable File

Coding → Compilation → Linking → Execution



1-18

1.4 What Is a Program Made of?

Common elements in programming languages:

- – Key Words 關鍵字，或保留字
- – Programmer-Defined Identifiers 識別符號，或名字
- – Operators 運算子
- – Punctuation 標點符號
- – Syntax 語法

1-19

Example Program

```
#include <iostream>
using namespace std;

int main()
{
   double num1 = 5,
          num2, sum;
   num2 = 12;

   sum = num1 + num2;
   cout << "The sum is " << sum;
   return 0;
}
```

1-20

Key Words

- Also known as reserved words
- Have a special meaning in C++
- Can not be used for another purpose
- Written using lowercase letters
- Examples in program (shown in green):

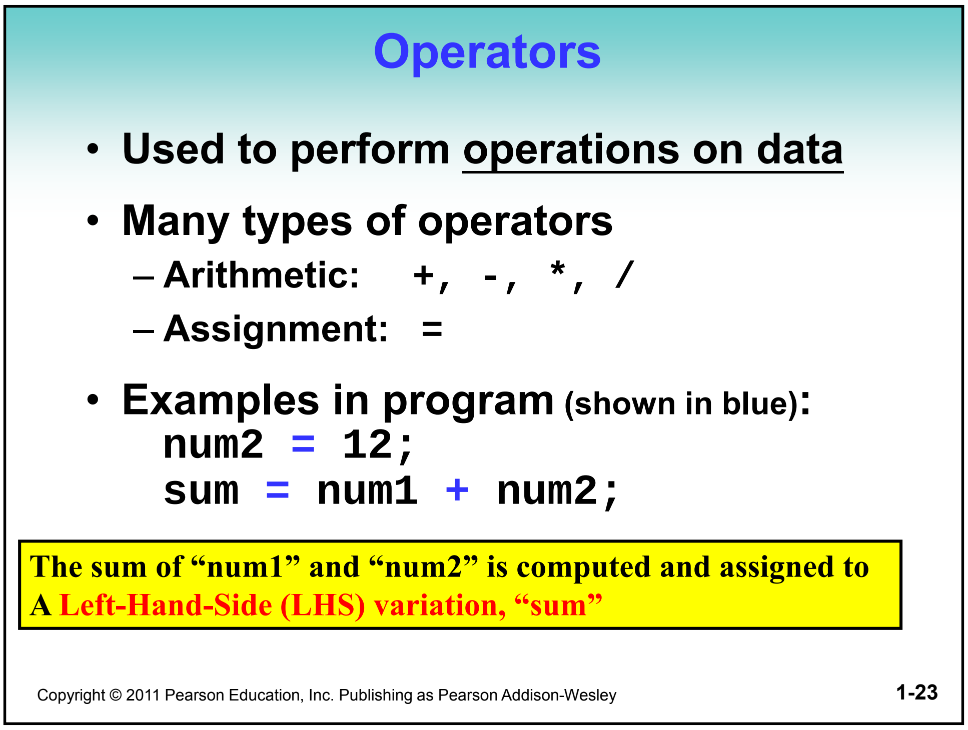
```
using namespace std;
int main()
```

Programmer-Defined Identifiers

- Names made up by the programmer
- Not part of the C++ language
- Used to represent various things, such as variables (memory locations)
- Example in program (shown in blue):

```
double num1
```

num1 is declared as a double-precision floating-point real number

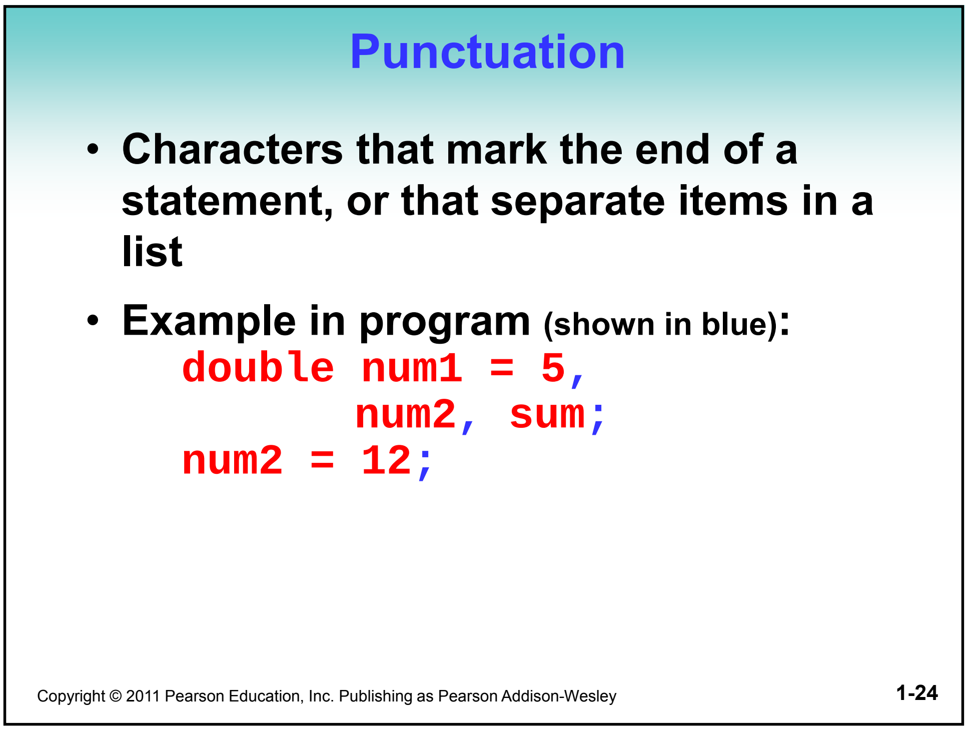


Operators

- Used to perform operations on data
- Many types of operators
  - Arithmetic: `+, -, *, /`
  - Assignment: `=`
- Examples in program (shown in blue):

```
num2 = 12;
sum = num1 + num2;
```

The sum of “num1” and “num2” is computed and assigned to A Left-Hand-Side (LHS) variation, “sum”



Punctuation

- Characters that mark the end of a statement, or that separate items in a list
- Example in program (shown in blue):

```
double num1 = 5,
       num2, sum;
num2 = 12;
```

Lines vs. Statements

In a source file,

A line is all of the characters entered before a carriage return.

Blank lines improve the readability of a program.

Here are four sample lines. Line 3 is blank:

```
double num1 = 5, num2, sum;
num2 = 12;

sum = num1 + num2;
```

Lines vs. Statements

In a source file,

A statement is an instruction to the computer to perform an action.

A statement may contain keywords, operators, programmer-defined identifiers, and punctuation.

A statement may fit in one line, or it may occupy multiple lines.

Here is a single statement that uses two lines:

```
double num1 = 5,
       num2, sum;
```

Variables

變數佔有記憶體中的一塊 (有位址) 的空間，可用於儲存資料

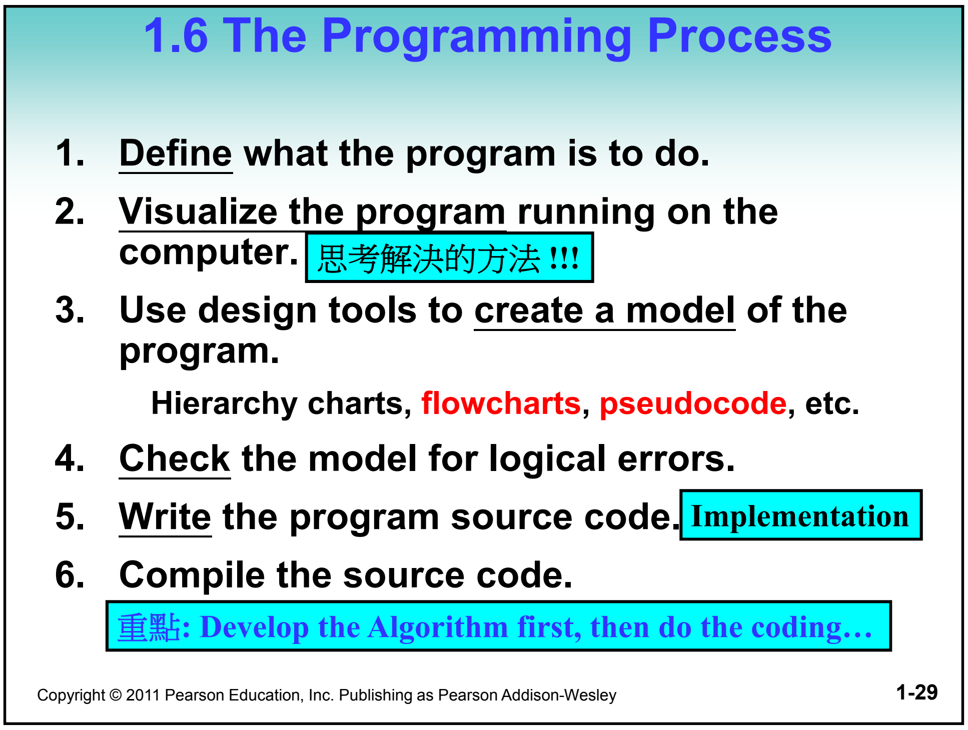
- A variable corresponds to a named location in computer memory (in RAM)
- It holds a piece of data
- It must be *defined* before it can be used
- Example variable definition:
  - `double num1;`

$$\begin{tikzcd}\texttt{double} \arrow[d] & \texttt{num1;} \arrow[d] \\ \text{type} & \text{name (or identifier)}\end{tikzcd}$$

1.5 Input, Processing, and Output

Three steps that many programs perform

1) Gather input data
   - from keyboard (Standard Input)
   - from files on disk drives (Input From Files)
2) Process the input data
3) Display the results as output
   - send it to the screen (Standard) or a printer
   - write it to a file (Output to Files)

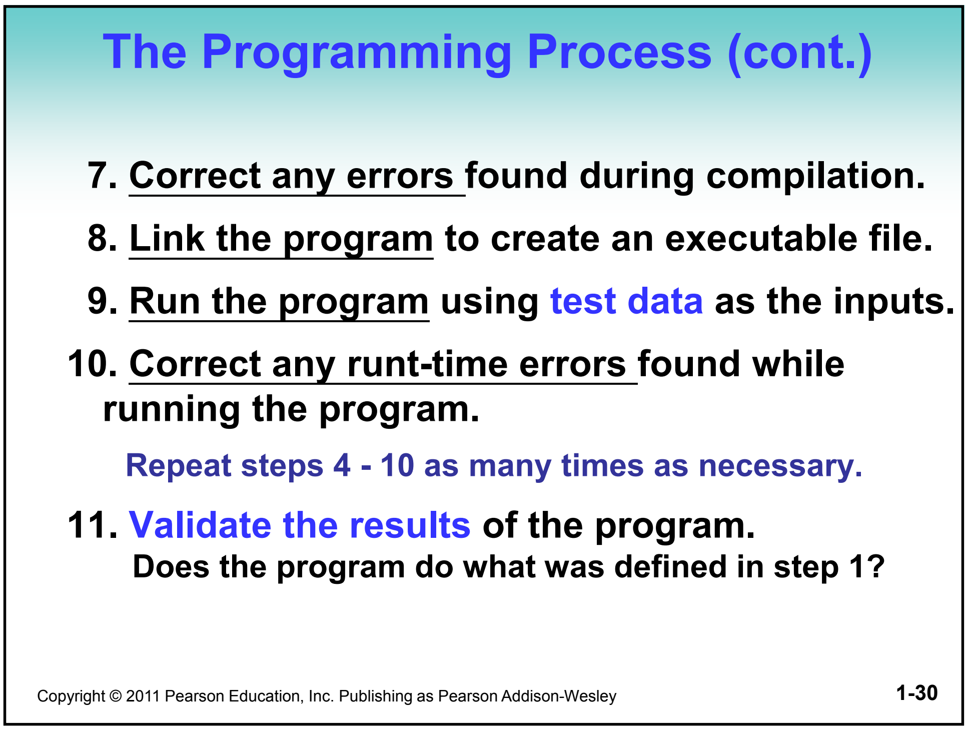


1.6 The Programming Process

1. Define what the program is to do.
2. Visualize the program running on the computer. 思考解決的方法 !!!
3. Use design tools to create a model of the program.

   Hierarchy charts, flowcharts, pseudocode, etc.
4. Check the model for logical errors.
5. Write the program source code. Implementation
6. Compile the source code.

重點: Develop the Algorithm first, then do the coding…



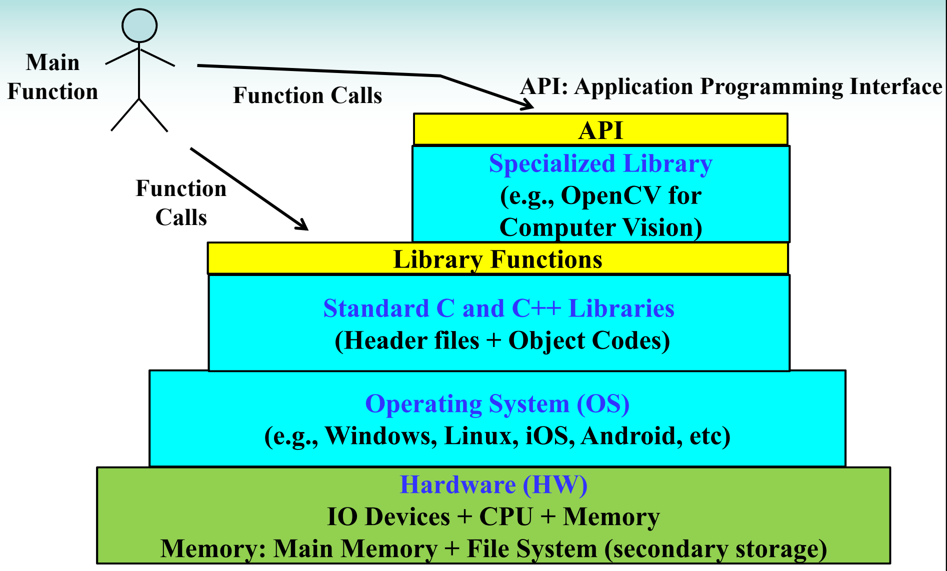
The Programming Process (cont.)

7. Correct any errors found during compilation.
8. Link the program to create an executable file.
9. Run the program using test data as the inputs.
10. Correct any runt-time errors found while running the program.

    Repeat steps 4 - 10 as many times as necessary.
11. Validate the results of the program.

    Does the program do what was defined in step 1?

Hierarchy from HW to SW


Main Function
Function Calls
API: Application Programming Interface
API
Specialized Library
(e.g., OpenCV for Computer Vision)
Function Calls
Library Functions
Standard C and C++ Libraries
(Header files + Object Codes)
Operating System (OS)
(e.g., Windows, Linux, iOS, Android, etc)
Hardware (HW)
IO Devices + CPU + Memory
Memory: Main Memory + File System (secondary storage)

1-31

Information Type

Number:

- Integer $(1010)_2$, $(999)_{10}$
- Fixed-point real number (e.g., 15.12)
- Floating-point real number (e.g., 1.5E129)

Text:

-'A' (character) 單引號
- "Programming in C" (string) 雙引號

Voice and Audio:

- Voice or Speech (lower-quality)
- Audio signal (high-fidelity)

Image and Video:

- Image is static
- video is dynamic (e.g., 30 frames per second)

1-32

Number System

Radix-r Number System

$$\boxed{\begin{aligned}(a_n \ldots a_1 a_0 \bullet a_{-1} \ldots a_{-m})_r &= a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \ldots. + a_2 \cdot r^2 + a_1 \cdot r^1 + a_0 \\ &+ a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} + \ldots. + a_{-m} \cdot r^{-m}\end{aligned}}$$

A fixed-point real number = (integer part) + (fractional part)

Examples:

$$(4021.2)_5 = 4 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 1 \times 5^0 + 2 \times 5^{-1} = (511.4)_{10}$$

$$\begin{aligned}(110101) &= 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^1 \\ &= 32 + 16 + 4 + 1 = (53)_{10}\end{aligned}$$

Octal & Hexadecimal Numbers

Converting a binary number into its octal form: (by grouping)

$$\begin{aligned}&(10 \quad 110 \quad 001 \quad 101 \quad 011 \quad \cdot \quad 111 \quad 100 \quad 000 \quad 110)_2 \\ &= (26153.7406)_8\end{aligned}$$

Converting a binary number into its hexadecimal form:

$$\begin{aligned}&(10 \quad 1100 \quad 0110 \quad 1011 \quad \cdot \quad 1111 \quad 0010)_2 \\ &= (2C6B.F2)_{16}\end{aligned}$$

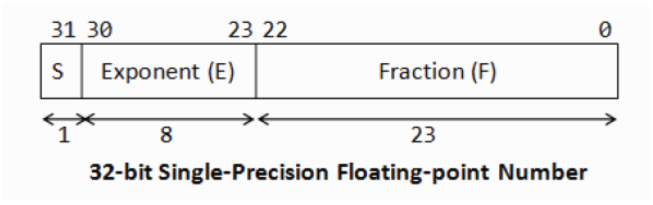
Hexadecimal digits:

A for 10	**B for 11**	**C for 12**
D for 13	**E for 14**	**F for 15**

IEEE-754 32-bit Single-Precision Floating-Point Numbers

In 32-bit single-precision floating-point representation:

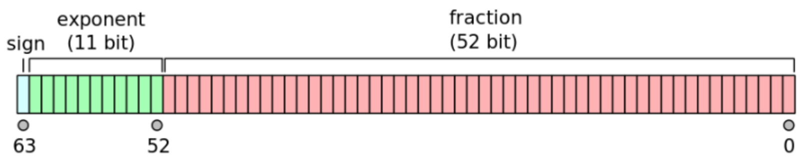
- The most significant bit is the *sign bit* (S), with 0 for positive numbers and 1 for negative numbers.
- The following 8 bits represent *exponent* (E).
- The remaining 23 bits represents *fraction* (F).



Example: 1 1000 0001 011 0000 0000 0000 0000 0000

There is a hidden 1 in the fraction
➔ So, the actual fraction is 1.F ➔ $(1.011)_2 = 1 + 1x2^{-2}+1x2^{-3} = (1.375)_{10}$
There is a bias of 127 in the exponent (which represents only a positive integer)
➔The actual exponent is E-127 ➔ $(10000001) – 127 = 129 – 127 = (2)_{10}$
The number represented is $-1.375x2^2 = (-5.5)_{10}$

IEEE-754 64-bit Single-Precision Floating-Point Numbers



The real value assumed by a given 64-bit double-precision datum with a given biased exponent *e* and a 52-bit fraction is

$$(-1)^{\text{sign}}(1.b_{51}b_{50}\ldots b_0)_2 \times 2^{e-1023}$$

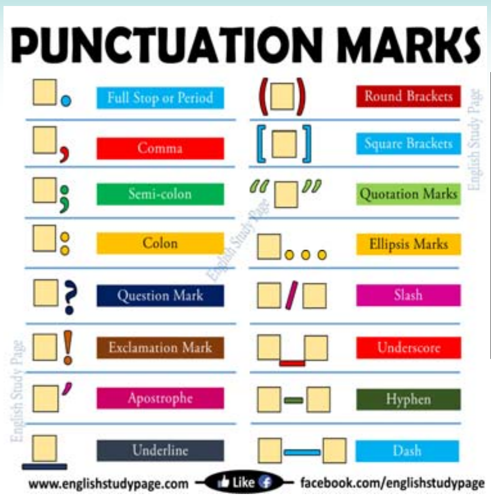
Useful Tips in Programming

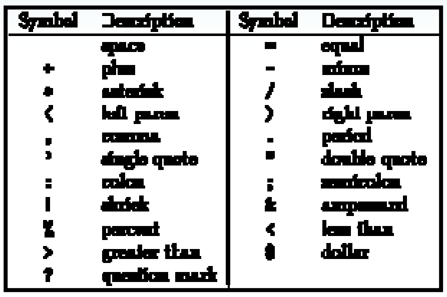
- Team Work
  - Conventions (e.g., how to name functions or variables)
  - Job Partitioning
  - Input Parsing and Data Processing Engine
- Data Structure Planning
  - Header Files and Global Variables
- Algorithm
- Compiler Options
  - -g for Debugging, -O for Optimization before releasing the code

Learning Roadmap to Become an Expert Program

- Introduction to Computer Program
- Data Structure
- Algorithm
- Multimedia Signal Processing (mathematical)
- Internet Programming (IoT programming)
- Machine Learning & Artificial Intelligence

Punctuation Marks & Special Symbols





Symbol	Description	Symbol	Description
	space	=	equal
+	plus	-	minus
*	asterisk	/	slash
(	left paren	)	right paren
,	comma	.	period
'	single quote	"	double quote
:	colon	;	semicolon
!	shriek	&	ampersand
%	percent	<	less than
>	greater than	$	dollar
?	question mark		

Chapter 2: Introduction to C++

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Topics

2.1 The Parts of a C++ Program
2.2 The `cout` Object
2.3 The `#include` Directive
2.4 Standard and Prestandard C++
2.5 Variables, Constants, and the Assignment Statement
2.6 Identifiers
2.7 Integer Data Types
2.8 The `char` Data Type

Topics (continued)

2.9 The C++ `string` Class

2.10 Floating-Point Data Types

2.11 The `bool` Data Type

2.12 Determining the Size of a Data Type

2.13 More on Variable Assignments and Initialization

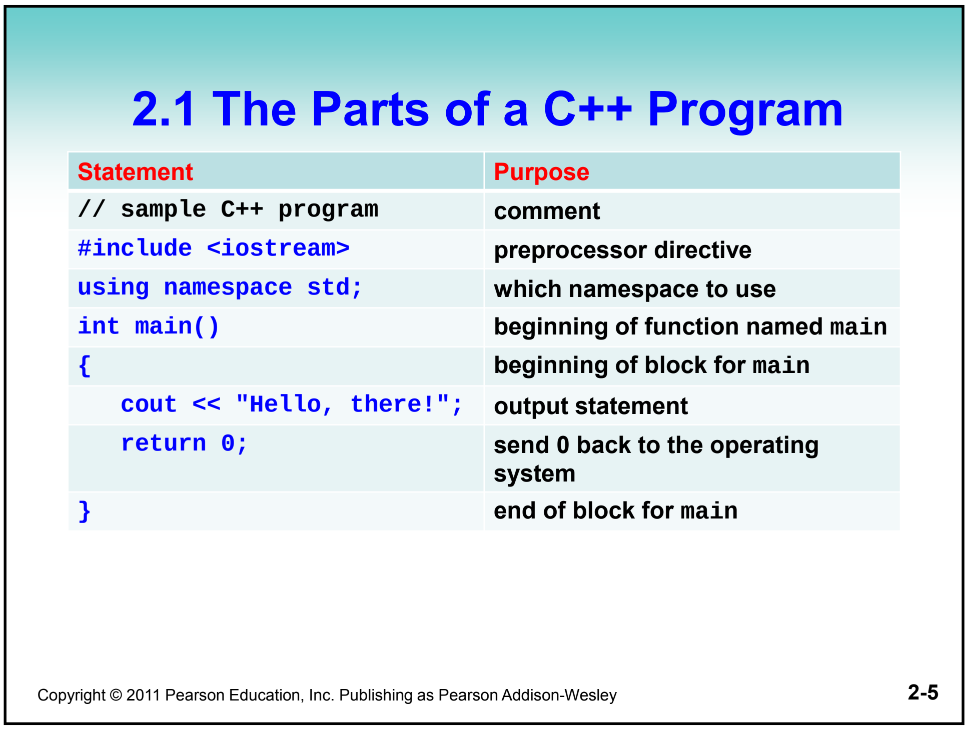
2.14 Scope

2.15 Arithmetic Operators

2.16 Comments

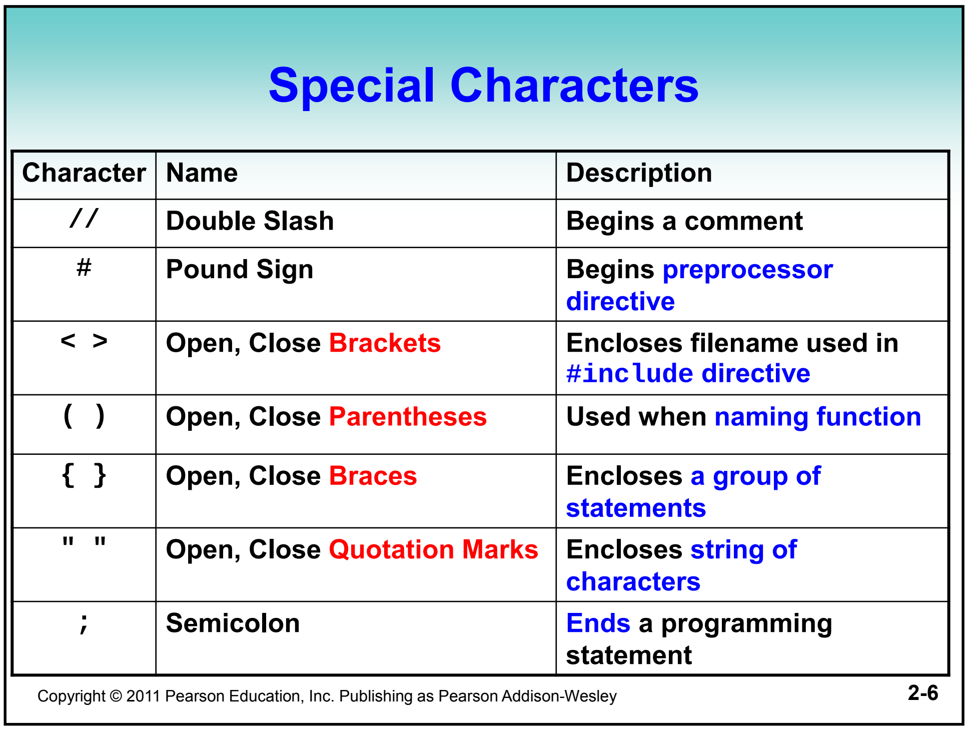
2.1 The Parts of a C++ Program

```
// sample C++ program            ←——— comment
#include <iostream>            ←——— preprocessor directive
using namespace std;          ←—— which namespace to use
int main()       ←———— beginning of function named main
{       ←———————————— beginning of block for  main
   cout << "Hello, there!"; ←—output statement
   return 0;  ←——————— send 0 back to operating system
}      ←——————————————— end of block for main
```



2.1 The Parts of a C++ Program

Statement	Purpose
`// sample C++ program`	comment
`#include <iostream>`	preprocessor directive
`using namespace std;`	which namespace to use
`int main()`	beginning of function named `main`
`{`	beginning of block for `main`
`   cout << "Hello, there!";`	output statement
`   return 0;`	send 0 back to the operating system
`}`	end of block for `main`



Special Characters

Character	Name	Description
`//`	Double Slash	Begins a comment
`#`	Pound Sign	Begins preprocessor directive
`< >`	Open, Close Brackets	Encloses filename used in #include directive
`( )`	Open, Close Parentheses	Used when naming function
`{ }`	Open, Close Braces	Encloses a group of statements
`" "`	Open, Close Quotation Marks	Encloses string of characters
`;`	Semicolon	Ends a programming statement

Important Details

- C++ is case-sensitive. Uppercase and lowercase characters are different characters. ‘`Main`’ is not the same as ‘`main`’.
- Every { must have a corresponding }, and vice-versa.

2.2 The cout Object

- Displays information on computer screen
- Use << to send information to cout

```
cout << "Hello, there!";
```

- Can use << to send multiple items to cout

```
cout << "Hello, " << "there!";
```

Or

```
cout << "Hello, ";
cout << "there!";
```

Starting a New Line

- To get multiple lines of output on screen
  - Use `endl`

    ```
    cout << "Hello, there!" << endl;
    ```

  - Use `\n` in an output string

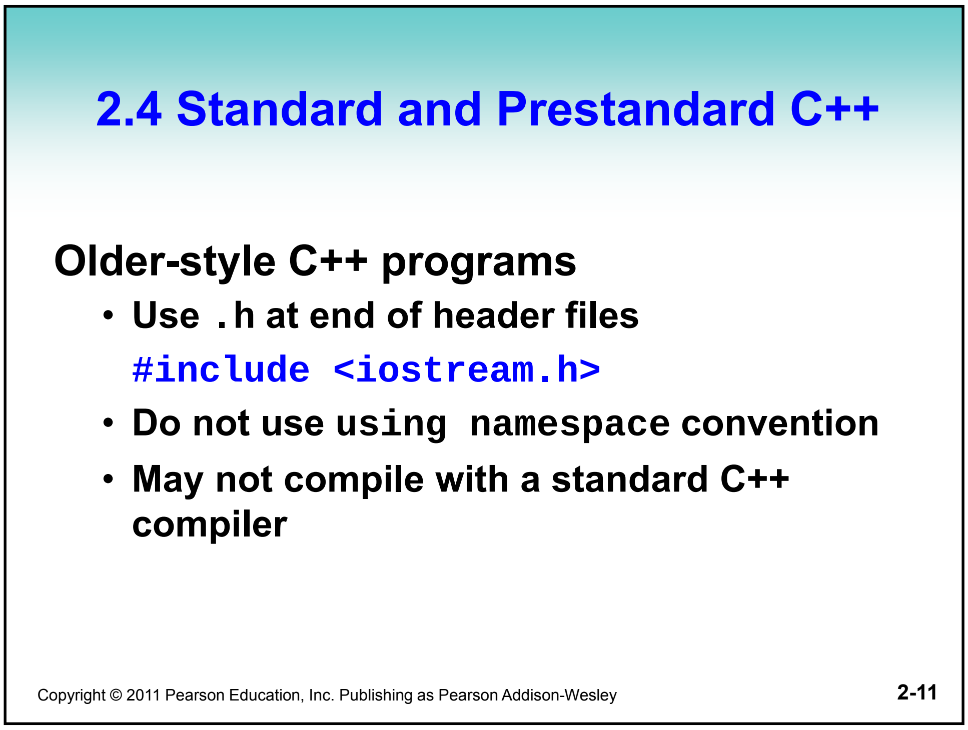
    ```
    cout << "Hello, there!\n";
    ```

2.3 The #include Directive

- Inserts the contents of another file into the program
- It is a **preprocessor directive**
  - Not part of the C++ language
  - Not seen by compiler
- Example:

  ```
  #include <iostream>
  ```

No ;
goes here

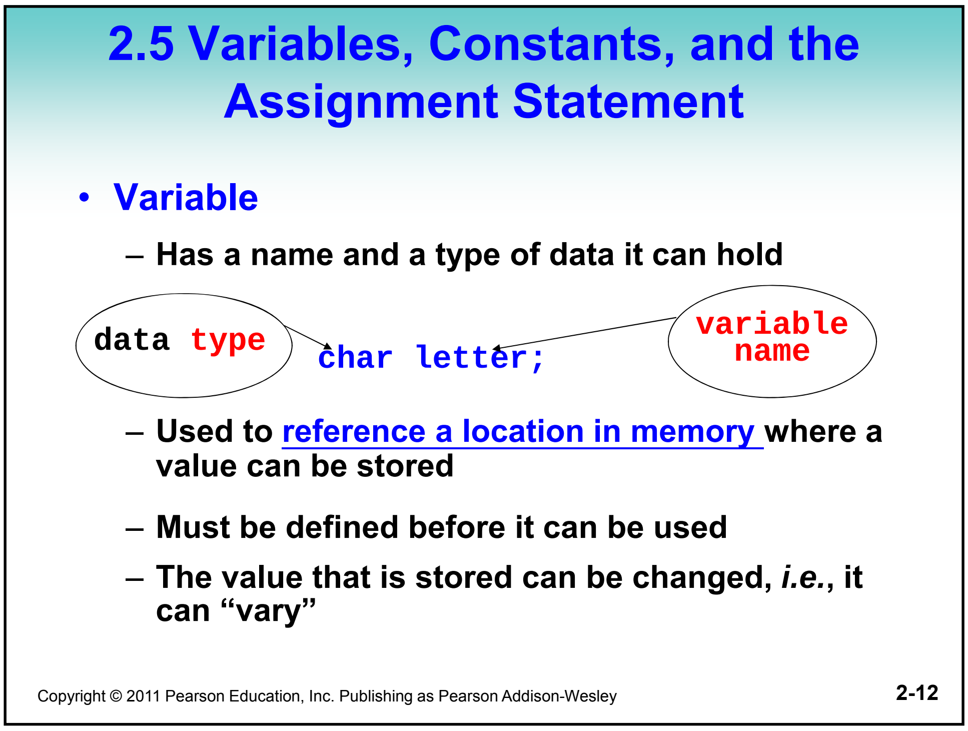


2.4 Standard and Prestandard C++

Older-style C++ programs

- Use `.h` at end of header files

  `#include <iostream.h>`
- Do not use `using namespace` convention
- May not compile with a standard C++ compiler



2.5 Variables, Constants, and the Assignment Statement

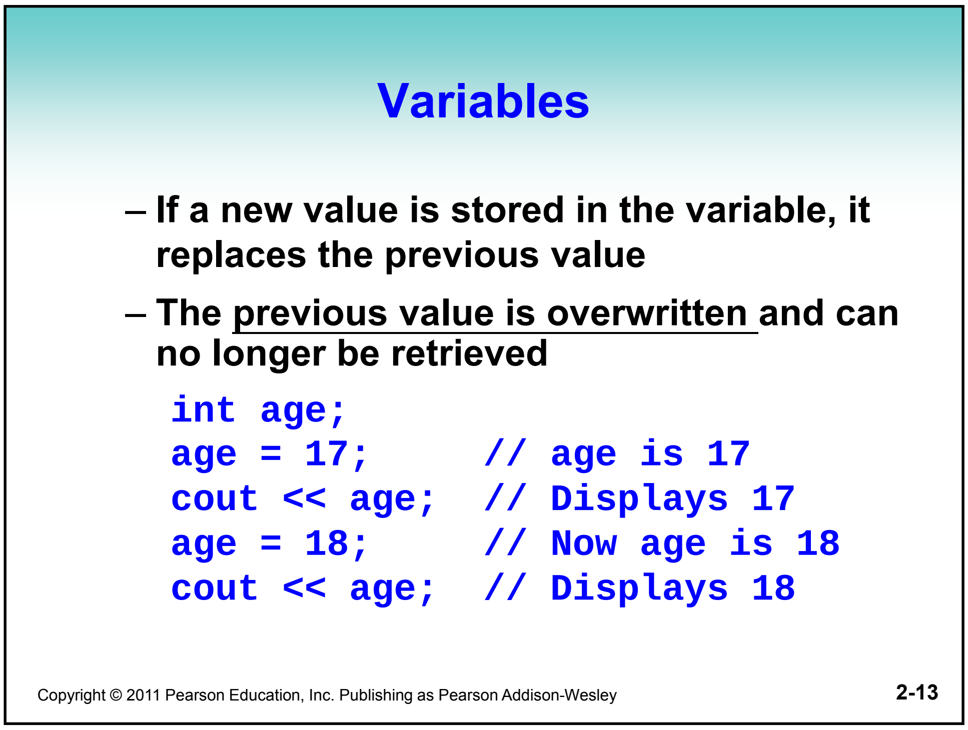
- Variable
  - Has a name and a type of data it can hold

data type

```
char letter;
```

variable name

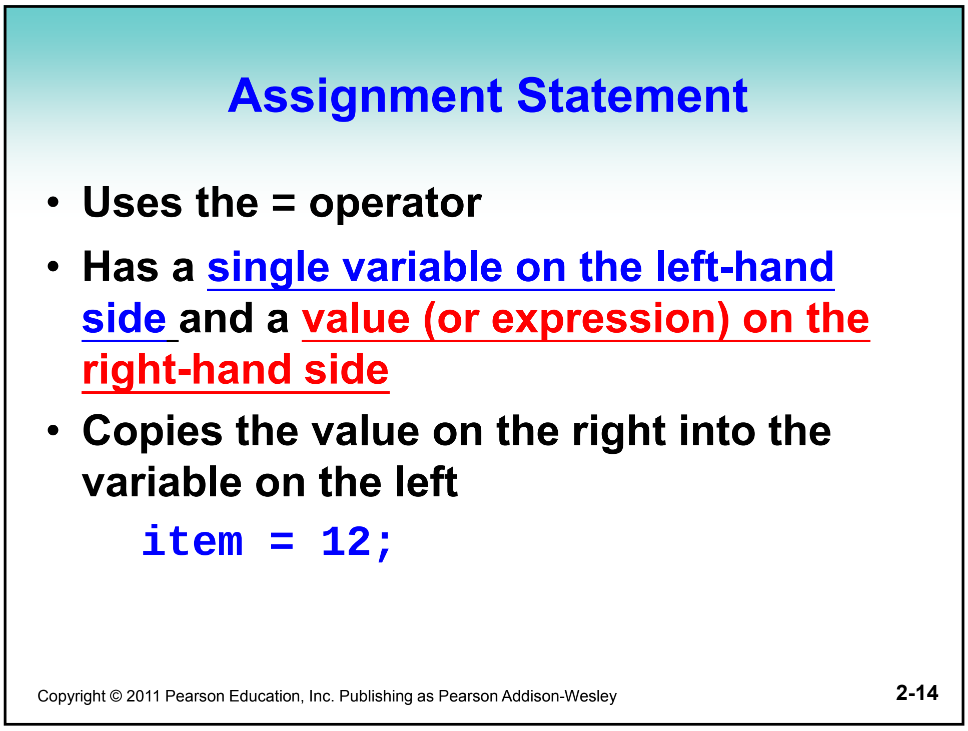
  - Used to reference a location in memory where a value can be stored
  - Must be defined before it can be used
  - The value that is stored can be changed, *i.e.*, it can "vary"



Variables

- If a new value is stored in the variable, it replaces the previous value
- The previous value is overwritten and can no longer be retrieved

```
int age;
age = 17;       // age is 17
cout << age;    // Displays 17
age = 18;       // Now age is 18
cout << age;    // Displays 18
```



Assignment Statement

- Uses the = operator
- Has a single variable on the left-hand side and a value (or expression) on the right-hand side
- Copies the value on the right into the variable on the left

```
item = 12;
```

Constants

- Constant
  - Data item whose value does not change during program execution
  - Is also called a literal

```
'A'      // character constant
"Hello"  // string literal
12       // integer constant
3.14     // floating-point constant
```

2.6 Identifiers

- Programmer-chosen names to represent parts of the program, such as variables
- Name should indicate the use of the identifier
- Cannot use C++ key words as identifiers
- Must begin with alphabetic character or _, followed by alphabetic, numeric, or _ . Alpha may be upper- or lowercase

Valid and Invalid Identifiers

IDENTIFIER	VALID?	REASON IF INVALID
`totalSales`	Yes	
`total_Sales`	Yes	
`total.Sales`	No	Cannot contain period
`4thQtrSales`	No	Cannot begin with digit
`totalSale$`	No	Cannot contain $

2.7 Integer Data Types

- Designed to hold whole numbers
- Can be `signed` or `unsigned`

  `12        -6        +3`

- Available in different sizes (*i.e.*, number of bytes): `short`, `int`, and `long`
- Size of `short` ≤ size of `int` ≤ size of `long`

Defining Variables

- Variables of the same type can be defined
  - In separate statements

```
int length;
int width;
```

  - In the same statement

```
int length,
    width;
```

- Variables of different types must be defined in separate statements

Integral Constants

- To store an integer constant in a long memory location, put ‘`L`’ at the end of the number: `1234L`
- Constants that begin with ‘`0`’ (zero) are octal, or base 8: `075`
- Constants that begin with ‘`0x`’ are hexadecimal, or base 16: `0x75A`

Hexadecimal digits: {0, 1, …, 9, A, B, C, D, E, F}
➔ to represent 0 to ’15’

2.8 The char Data Type

- Used to hold single character or very small integer values
- Usually occupies 1 byte of memory
- A numeric code (e.g., ASCII code) representing the character is stored in memory

SOURCE CODE
```
char letter = 'C';
```
MEMORY
```
letter
```
$\boxed{67}$

ASCII stands for American Standard Code for Information Interchange

String Constant

- Can be stored a series of characters in consecutive memory locations

```
"Hello"
```

- Stored with the null terminator, `'\0'`, at the end

H	e	l	l	o	\0

- Is composed of characters between the " "

A character or a string constant?

- A character constant is a single character, enclosed in single quotes:
  `'C'`
- A string constant is a sequence of characters enclosed in double quotes:
  `"Hello, there!"`
- A single character in double quotes is a string constant, not a character constant:
  `"C"` ➔ `character or string?`

2.9 The C++ `string` Class

- Must `#include <string>` to create and use string objects
- Can define `string` variables in programs
  `string name;`
- Can assign values to string variables with the assignment operator
  `name = "George";`
- Can display them with cout
  `cout << name;`

2.10 Floating-Point Data Types

- Designed to hold real numbers

  `12.45`   `-3.8`
- Stored in a form similar to scientific notation
- Numbers are all signed
- Available in different sizes (number of bytes): `float`, `double`, and `long double`
- Size of `float` ≤ size of `double` ≤ size of `long double`

Floating-point Constants

- Can be displayed in a program
  - Fixed point (decimal) notation:

    `31.4159`   `0.0000625`
  - E notation:

    `3.14159E1`   `6.25e-5`
- Are `double` by default
- Can be forced to be float `3.14159F` or long double `0.0000625L`

Assigning Floating-point Values to Integer Variables

If a floating-point value is assigned to an integer variable

- The fractional part will be truncated (*i.e.*, "chopped off" and discarded)
- The value is not rounded

```
int rainfall = 3.88;
cout << rainfall;  // Displays 3
```

2.11 The `bool` Data Type

- Represents values that are `true` or `false`
- `bool` values are stored as short integers
- `false` is represented by 0, `true` by 1

```
bool allDone = true;       allDone   finished
bool finished = false;      [ 1 ]     [ 0 ]
```

2.12 Determining the Size of a Data Type

The `sizeof` operator gives the size of any data type or variable

A byte has 8 bits

```
double amount;
cout << "A float is stored in "
     << sizeof(float) << " bytes\n";
cout << "Variable amount is stored in "
     << sizeof(amount) << " bytes\n";
```

2.13 More on Variable Assignments and Initialization

- Assigning a value to a variable
  - Assigns a value to a previously created variable
  - A single variable name must appear on left side of the = symbol

```
int size;
size = 5;    // legal
5 = size;    // not legal
```

Left-Hand Side (LHS) Should be a variable

Variable Assignment vs. Initialization

- Initializing a variable
  - Gives an initial value to a variable at the time it is created
  - Can initialize some or all variables of definition

```
int length = 12;
int width = 7, height = 5, area;
```

2.14 Scope

- The scope of a variable is that part of the program where the variable may be used
- A variable cannot be used before it is defined

```
int a;
cin >> a;    // legal
cin >> b;    // illegal
int b;
```

2.15 Arithmetic Operators

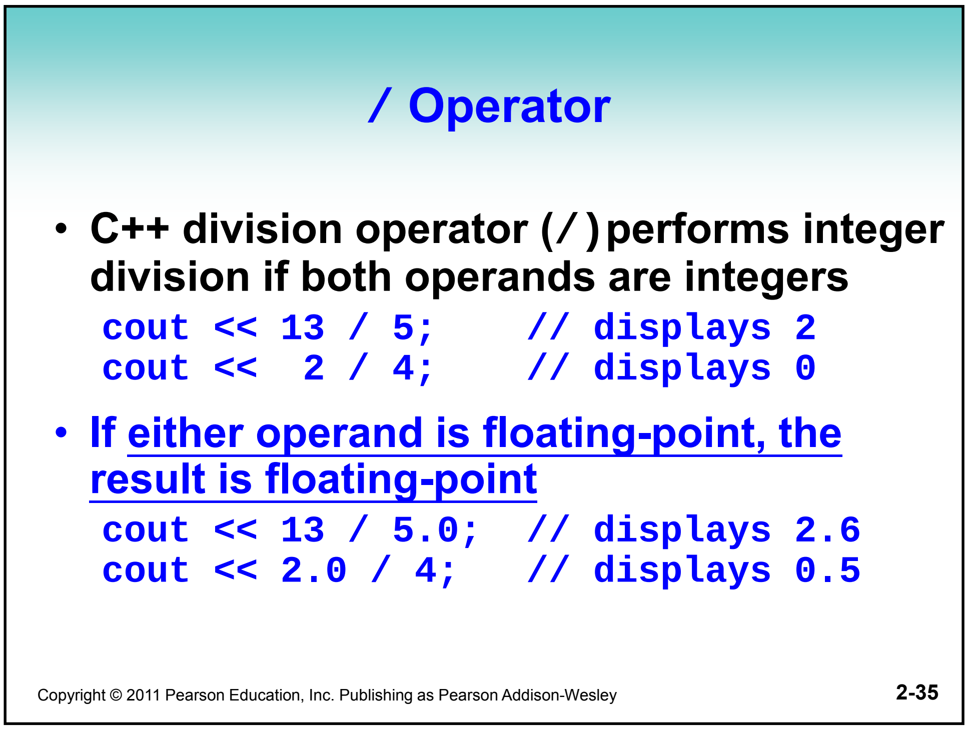
- Used for performing numeric calculations
- C++ has unary, binary, and ternary operators
  - unary (1 operand) `-5`
  - binary (2 operands) `13 - 7`
  - ternary (3 operands) `exp1 ? exp2 : exp3`

條件運算式: **If exp1 is true, 結果為 exp2, 反之結果為 exp3**

Binary Arithmetic Operators

SYMBOL	OPERATION	EXAMPLE	ans
`+`	addition	`ans = 7 + 3;`	`10`
`-`	subtraction	`ans = 7 - 3;`	`4`
`*`	multiplication	`ans = 7 * 3;`	`21`
`/`	division	`ans = 7 / 3;`	`2`
`%`	modulus	`ans = 7 % 3;`	`1`

% 為取餘數運算



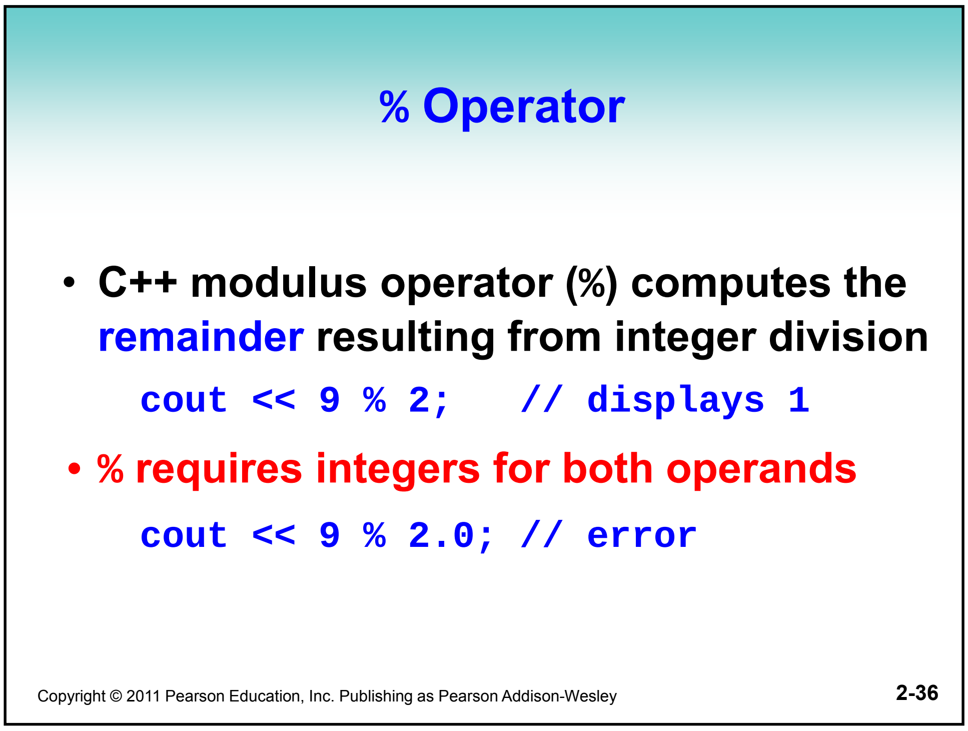
/ Operator

- C++ division operator (`/`) performs integer division if both operands are integers

```
cout << 13 / 5;     // displays 2
cout <<  2 / 4;     // displays 0
```

- If <u>either operand is floating-point, the result is floating-point</u>

```
cout << 13 / 5.0;  // displays 2.6
cout << 2.0 / 4;   // displays 0.5
```



% Operator

- C++ modulus operator (%) computes the remainder resulting from integer division

```
cout << 9 % 2;    // displays 1
```

- % requires integers for both operands

```
cout << 9 % 2.0; // error
```

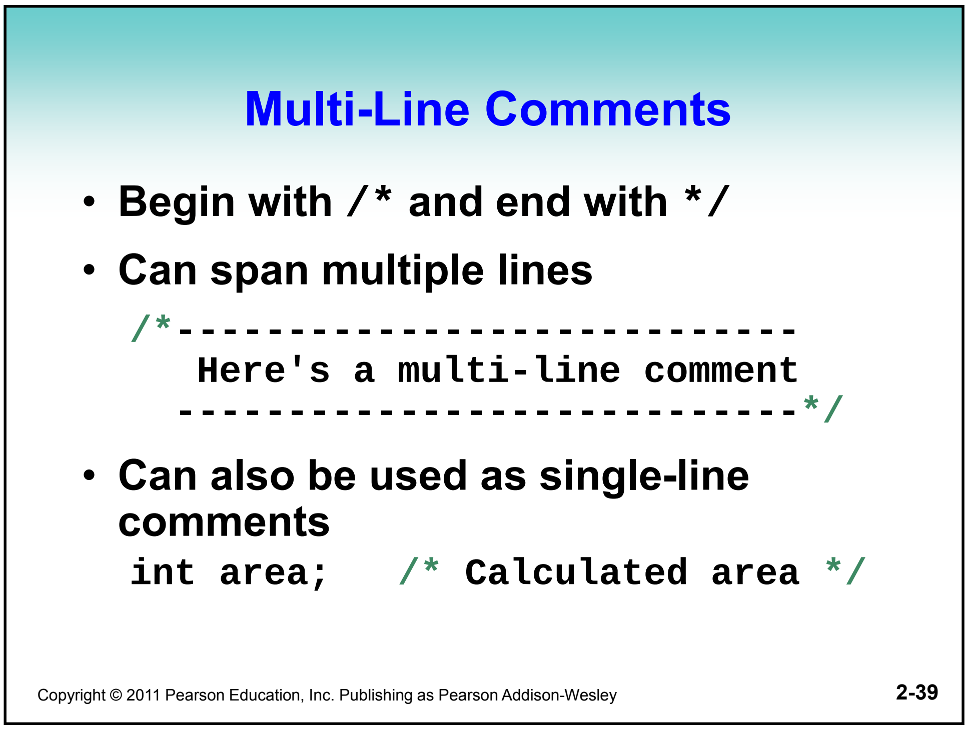
2.16 Comments

- Are used to document parts of a program
- Are written for persons reading the source code of the program
  - Indicate the purpose of the program
  - Describe the use of variables
  - Explain complex sections of code
- Are ignored by the compiler

Single-Line Comments

- Begin with // through to the end of line

```
int length = 12; // length in inches
int width = 15;  // width in inches
int area;        // calculated area

// Calculate rectangle area
area = length * width;
```



Multi-Line Comments

- Begin with `/*` and end with `*/`
- Can span multiple lines

```
/*----------------------------
   Here's a multi-line comment
  ----------------------------*/
```

- Can also be used as single-line comments

```
int area;   /* Calculated area */
```

Chapter 3: Expressions and Interactivity

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Topics

3.1 The `cin` Object
3.2 Mathematical Expressions
3.3 Implicit Type Conversion
3.4 Explicit Type Conversion
3.5 Overflow and Underflow
3.6 Named Constants

Topics (continued)

3.7 Multiple and Combined Assignment
3.8 Formatting Output
3.9 Working with Characters and String Objects
3.10 Using C-Strings
3.11 More Mathematical Library Functions
3.12 Introduction to Files

3.1 The `cin` Object

- Standard input object
- Like `cout`, requires `iostream` file
- Used to read input from keyboard
- Often used with `cout` to display a user prompt first
- Data is retrieved from `cin` with `>>`
- Input data is stored in one or more variables

The `cin` Object

- User input goes from **keyboard** to the **input buffer**, where it is stored as characters
- `cin` converts the data to the type that matches the variable

```
int height;
cout << "How tall is the room? ";
cin  >> height;
```

The `cin` Object

- Can be used to input multiple values

```
cin >> height >> width;
```

- **Multiple values from keyboard** must be separated by spaces or [Enter]
- Must press [Enter] after typing last value
- Multiple values need not all be of the same type
- **Order is important**; first value entered is stored in first variable, etc.

3.2 Mathematical Expressions

- An expression can be a constant, a variable, or a combination of constants and variables combined with operators
- Can create complex expressions using multiple mathematical operators
- Examples of mathematical expressions:
- `2`
- `height`
- `a + b / c`

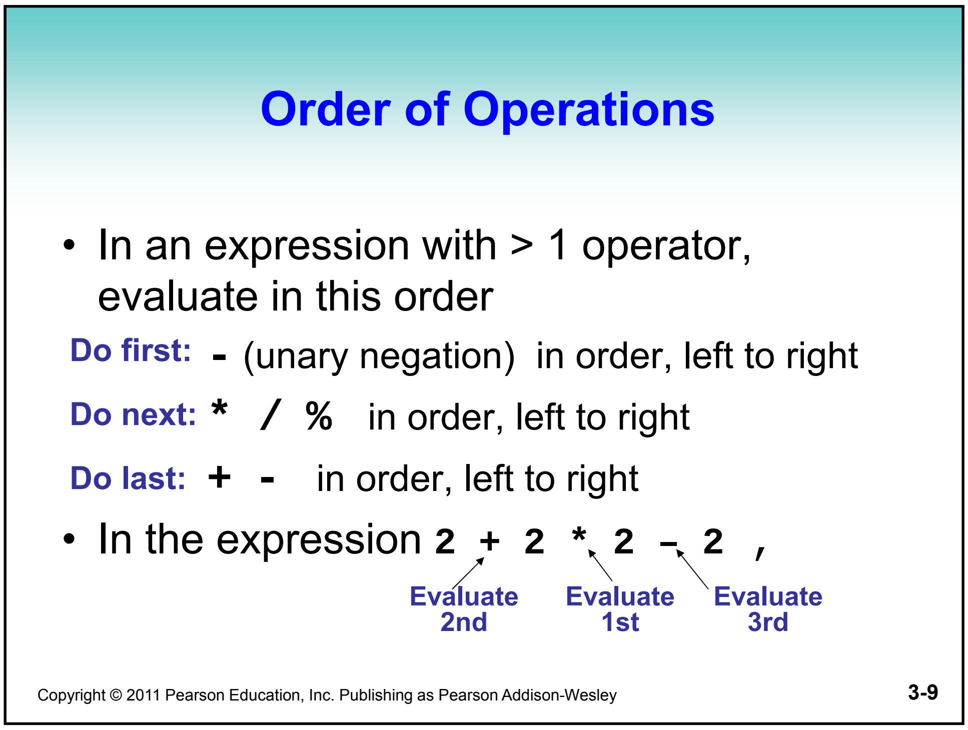
Using Mathematical Expressions

- Can be used in assignment statements, with `cout`, and in other types of statements
- Examples:

This is an expression

```
area = 2 * PI * radius;
cout << "border is: " << (2*(l+w));
```

These are expressions



Order of Operations

- In an expression with > 1 operator, evaluate in this order

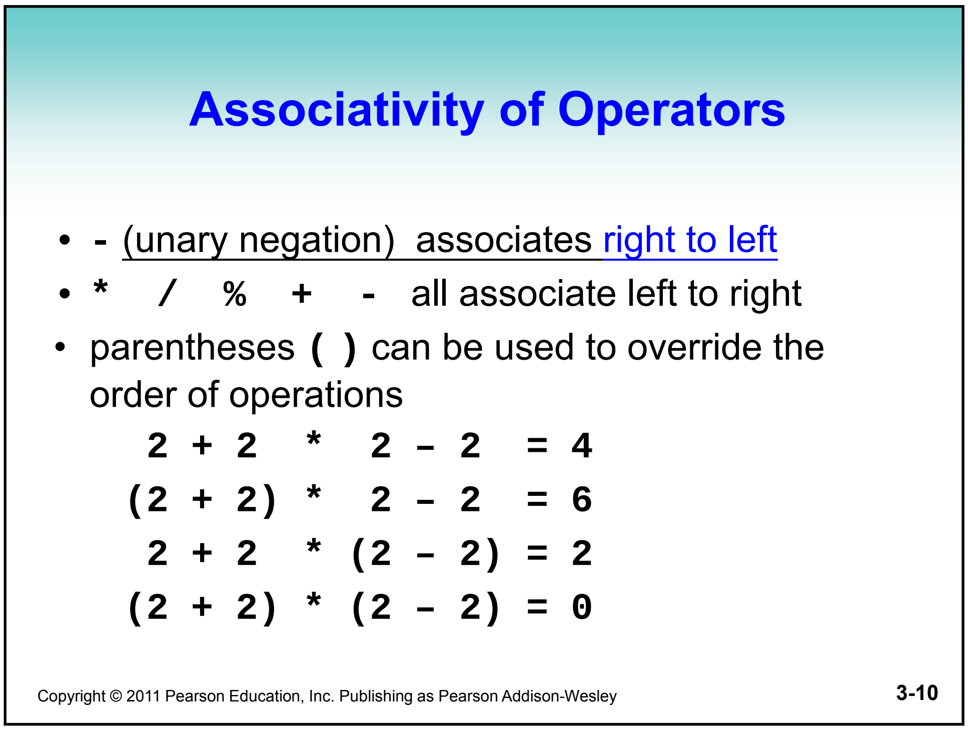
Do first: `-` (unary negation) in order, left to right

Do next: `*  /  %` in order, left to right

Do last: `+  -` in order, left to right

- In the expression `2 + 2 * 2 – 2 ,`

Evaluate 2nd (`+`) Evaluate 1st (`*`) Evaluate 3rd (`–`)



Associativity of Operators

- `-` (unary negation) associates right to left
- `*  /  %  +  -` all associate left to right
- parentheses `( )` can be used to override the order of operations

```
 2 + 2  *  2 – 2  = 4
(2 + 2) *  2 – 2  = 6
 2 + 2  * (2 – 2) = 2
(2 + 2) * (2 – 2) = 0
```

Algebraic Expressions

- Multiplication requires an operator

  Area = *lw* is written as `Area = l * w;`
- There is no exponentiation operator

  $Area = s^2$ is written as `Area = pow(s, 2);`

  (note: pow requires the `cmath` header file)
- Parentheses may be needed to maintain order of operations

  $$m = \frac{y_2 - y_1}{x_2 - x_1}$$ is written as

  `m = (y2-y1)/(x2-x1);`

3.3 Implicit Type Conversion

- Operations need to be performed between operands of the same type
- If not of the same type, C++ will automatically convert one to be the type of the other
- This can impact the results of calculations

Hierarchy of Data Types

- Highest
- Lowest

```
long double
double
float
unsigned long
long
unsigned int
int
unsigned short
short
char
```

- Ranked by largest number they can hold

Type Coercion

- **Coercion**: **automatic conversion** of an operand to another data type
- **Promotion**: converts to a <u>higher type</u>
- **Demotion**: converts to a <u>lower type</u>

Coercion Rules

1) `char`, `short`, `unsigned short` are automatically promoted to `int`
2) When operating on values of different data types, the lower one is promoted to the type of the higher one.
3) When using the = operator, the type of expression on right will be converted to the type of variable on the left

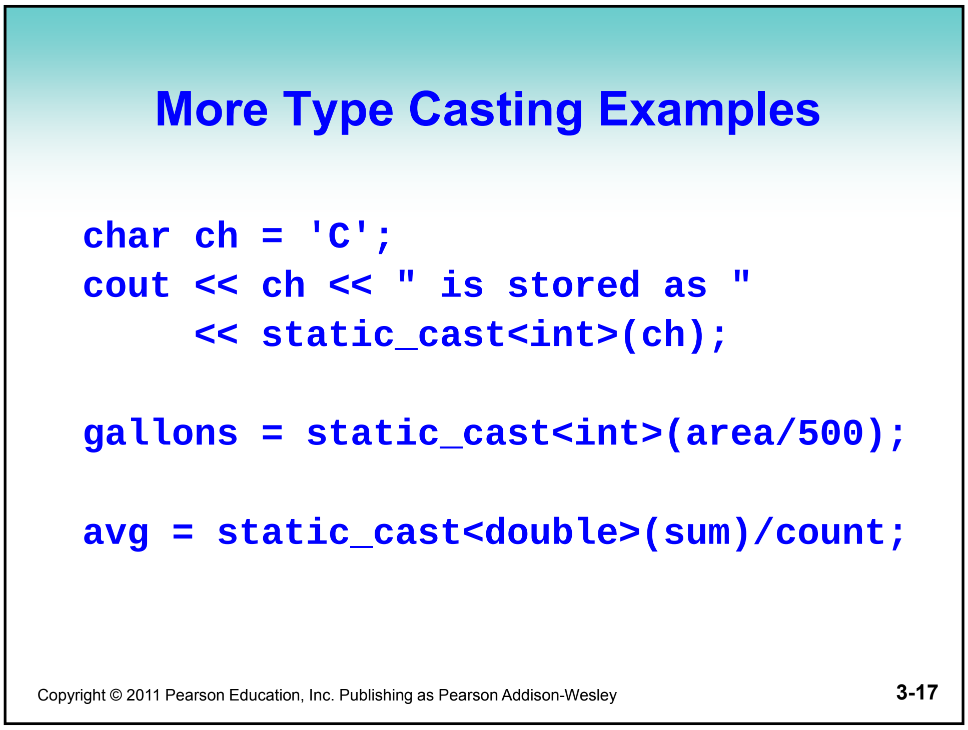
3.4 Explicit Type Conversion

- Also called type casting
- Used for manual data type conversion
- Format

```
    static_cast<type>(expression)
```

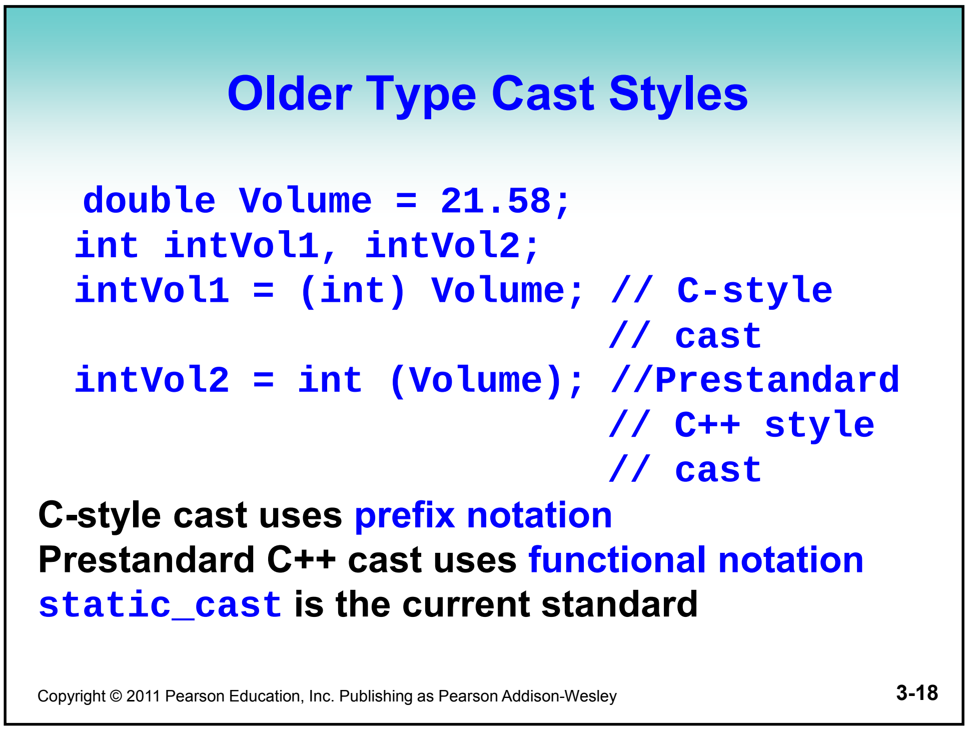
- Example:

```
cout << static_cast<char>(65);
                        // Displays 'A'
```



More Type Casting Examples

```
char ch = 'C';
cout << ch << " is stored as "
     << static_cast<int>(ch);

gallons = static_cast<int>(area/500);

avg = static_cast<double>(sum)/count;
```



Older Type Cast Styles

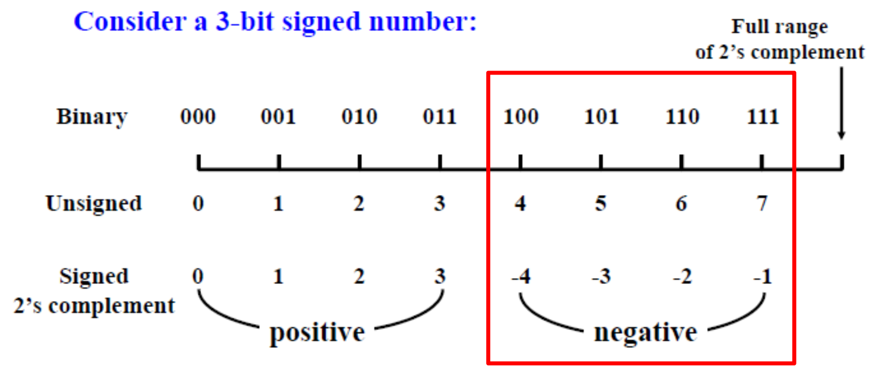
```
double Volume = 21.58;
int intVol1, intVol2;
intVol1 = (int) Volume; // C-style
                        // cast
intVol2 = int (Volume); //Prestandard
                        // C++ style
                        // cast
```

C-style cast uses prefix notation
Prestandard C++ cast uses functional notation
`static_cast` is the current standard

3.5 Overflow and Underflow

- Occurs when assigning a value that is too large (overflow) or too small (underflow) to be held in a variable
- The variable contains a value that is 'wrapped around' the set of possible values

Range of Signed Number



The MSB (Most Significant Bit) denotes the Sign

Overflow Example

```
// Create a short int initialized to
// the largest value it can hold
short int num = 32767;

cout << num;     // Displays 32767
num = num + 1;
cout << num;     // Displays -32768
```

Range of a 16-bit signed integer: [-2^{15}, $2^{15}-1$], $2^{15}=32768$

Handling Overflow and Underflow

Different systems handle the problem differently. They may

- display a warning / error message
- display a dialog box and ask what to do
- stop the program
- continue execution with the incorrect value

3.6 Named Constants

- Also called constant variables
- Variables whose content cannot be changed during program execution
- Used for representing constant values with descriptive names

```
const double TAX_RATE = 0.0675;
const int NUM_STATES = 50;
```

- Often named in uppercase letters

const vs. #define

#define

- C-style of naming constants

```
#define NUM_STATES 50
```

no ;
goes here

- Interpreted by pre-processor rather than compiler
- Does not occupy a memory location like a constant variable defined with const
- Instead, causes a text substitution to occur. In above example, every occurrence in program of NUM_STATES will be replaced by 50

3.7 Multiple and Combined Assignment

- The assignment operator (=) can be used more than 1 time in an expression

```
x = y = z = 5;
```

- Associates right to left

```
x = (y = (z = 5));
```

Done 3rd   Done 2nd   Done 1st

Combined Assignment

- Applies an arithmetic operation to a variable and assigns the result as the new value of that variable
- Operators: +=   -=   *=   /=   %=
- Example:
- `sum += amt;` is short for `sum = sum + amt;`

More Examples

```
x += 5;   means   x = x + 5;
x -= 5;   means   x = x – 5;
x *= 5;   means   x = x * 5;
x /= 5;   means   x = x / 5;
x %= 5;   means   x = x % 5;
```

The right hand side is evaluated before the combined assignment operation is done.

```
x *= a + b;  means  x = x * (a + b);
```

3.8 Formatting Output

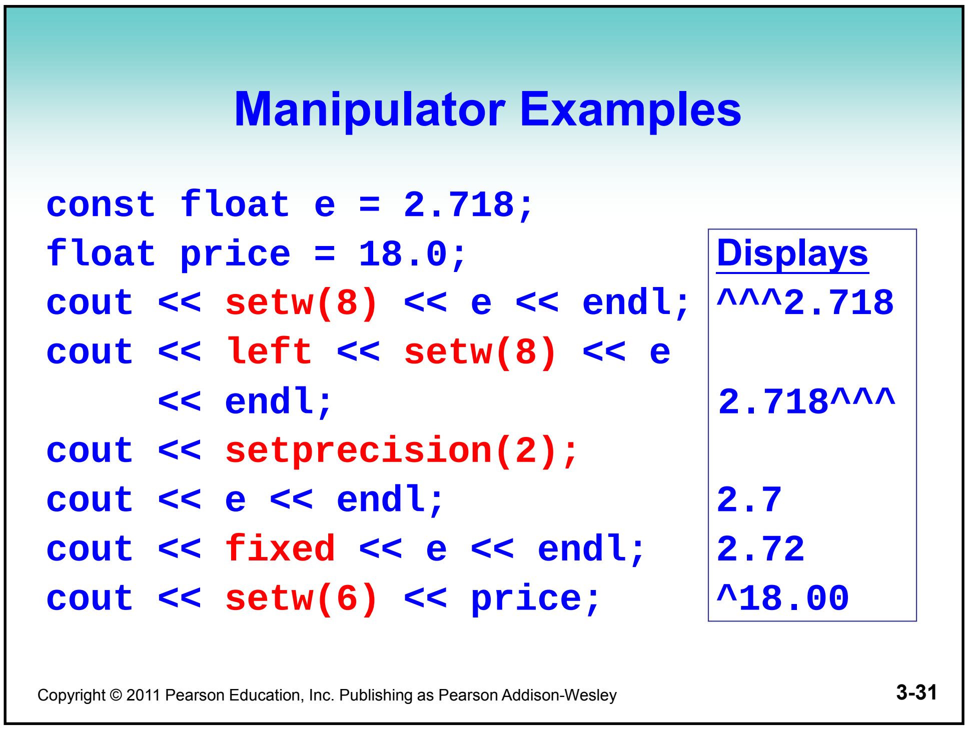
- **Can control how output displays for numeric and string data**
  - size
  - position
  - number of digits
- **Requires `iomanip` header file**

Stream Manipulators

- Used to control features of an output field
- Some affect just the next value displayed
  - `setw(x)`: Print in a field at least x spaces wide. Use more spaces if specified field width is not big enough.

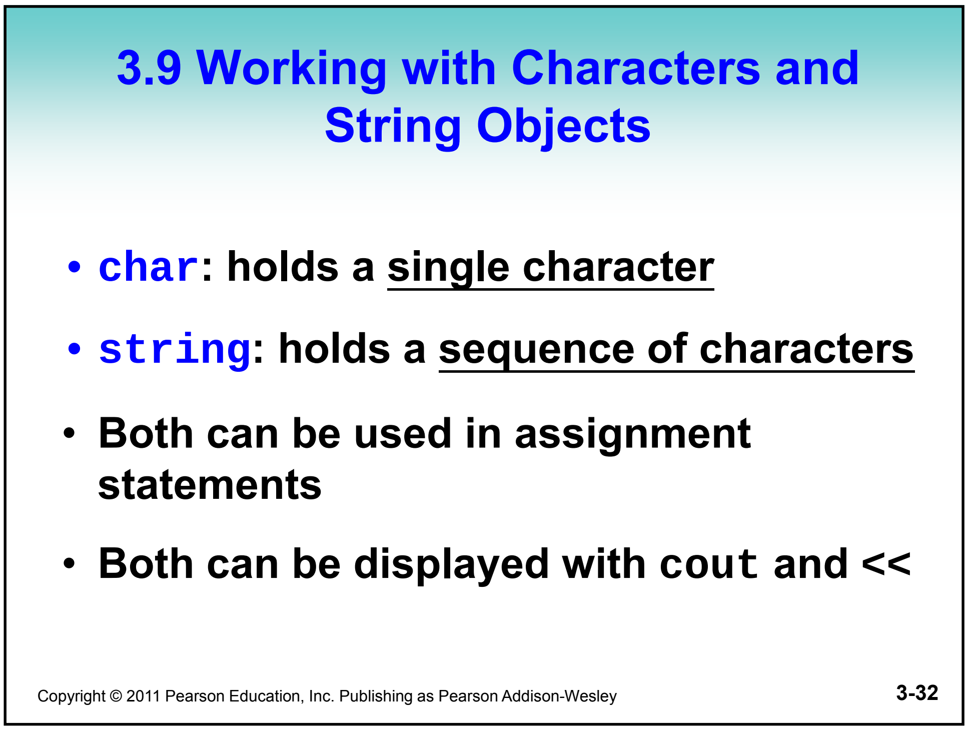
Stream Manipulators

- Some affect values until changed again
  - `fixed`: Use decimal notation (not E-notation) for floating-point values.
  - `setprecision(x)`:
    - When used with `fixed`, print floating-point value using x digits after the decimal.
    - Without `fixed`, print floating-point value using x significant digits.
  - `showpoint`: Always print decimal for floating-point values.
  - `left, right`: left-, right justification of value



Manipulator Examples

```
const float e = 2.718;
float price = 18.0;                  Displays
cout << setw(8) << e << endl;        ^^^2.718
cout << left << setw(8) << e
     << endl;                        2.718^^^
cout << setprecision(2);
cout << e << endl;                   2.7
cout << fixed << e << endl;          2.72
cout << setw(6) << price;            ^18.00
```



3.9 Working with Characters and String Objects

- `char`: holds a single character
- `string`: holds a sequence of characters
- Both can be used in assignment statements
- Both can be displayed with `cout` and `<<`

String Input

Reading in a string object

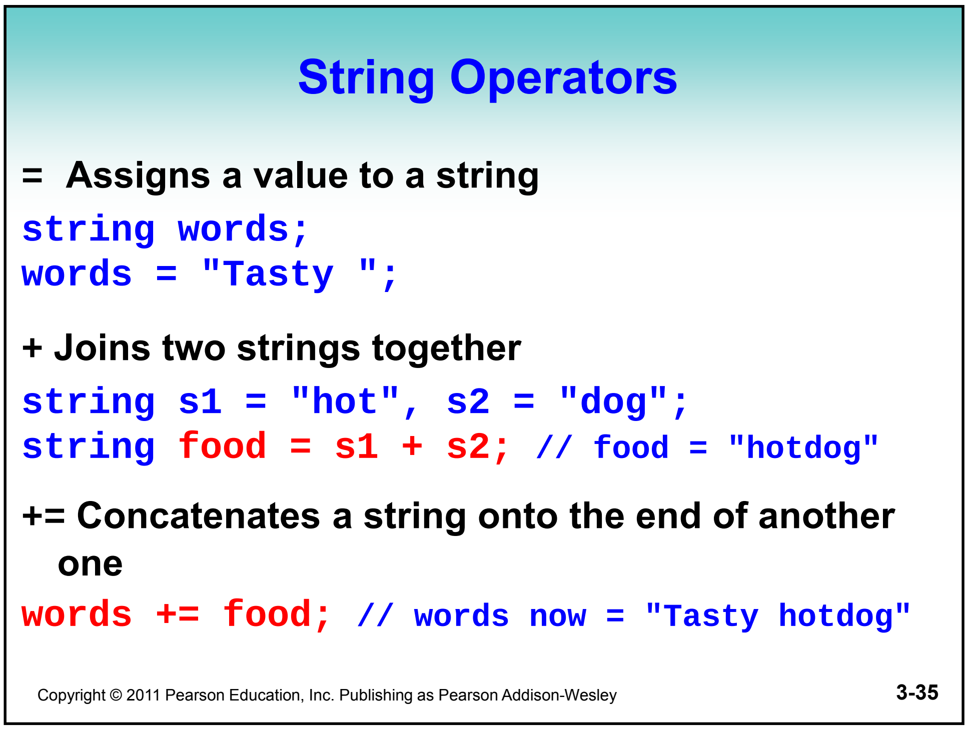
```
string str;
cin >> str;             // Reads in a string
                        // with no blanks
getline(cin, str);      // Reads in a string
                        // that may contain
                        // blanks
```

Character Input

Reading in a character

```
char ch;
cin >> ch;       // Reads in any non-blank char
cin.get(ch);     // Reads in any char
cin.ignore();    // Skips over next char in
                 // the input buffer
```

註: Object = Data + Member Functions
Use “dot operator” to call a member function associated with an object



String Operators

= Assigns a value to a string

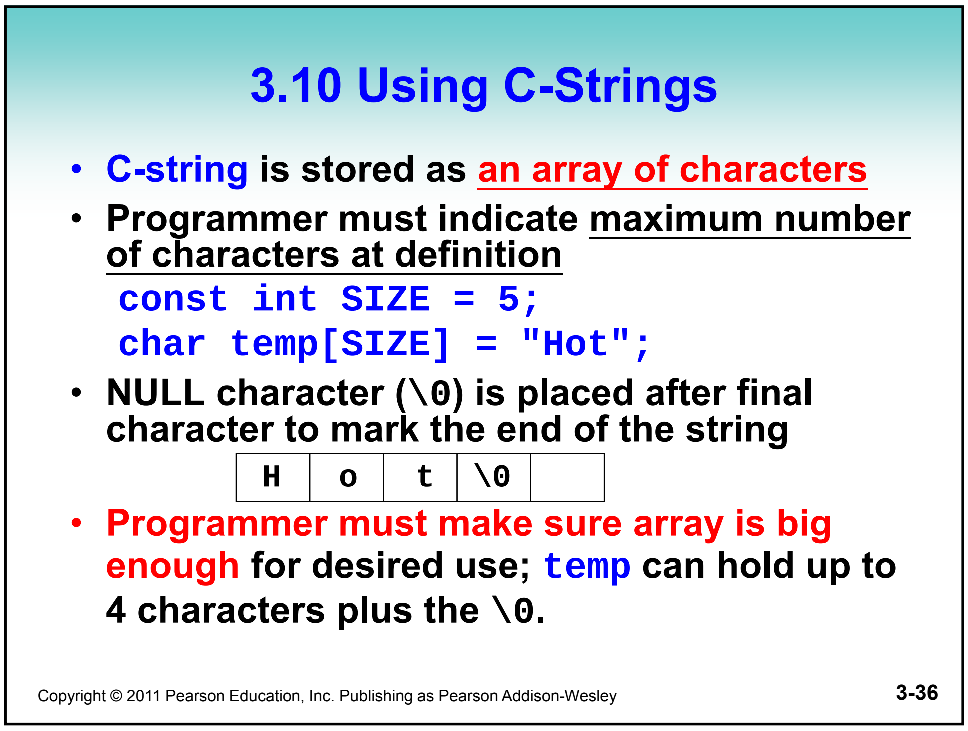
```
string words;
words = "Tasty ";
```

+ Joins two strings together

```
string s1 = "hot", s2 = "dog";
string food = s1 + s2; // food = "hotdog"
```

+= Concatenates a string onto the end of another one

```
words += food; // words now = "Tasty hotdog"
```



3.10 Using C-Strings

- C-string is stored as <u>an array of characters</u>
- Programmer must indicate <u>maximum number of characters at definition</u>

  ```
  const int SIZE = 5;
  char temp[SIZE] = "Hot";
  ```

- NULL character (`\0`) is placed after final character to mark the end of the string

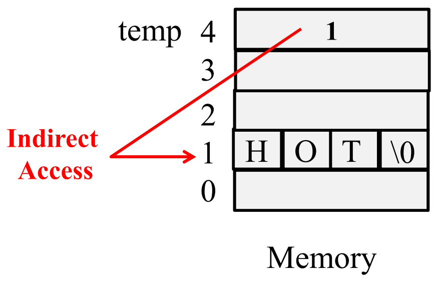
H	o	t	\0	

- Programmer must make sure array is big enough for desired use; `temp` can hold up to 4 characters plus the `\0`.

Array Name is “Pointer Type of Variable”

Array name is a “pointer type of variable” itself, containing the address to a large chunk of data in the memory

char temp[4] = “HOT”;



The char array is located in a chunk of consecutive data starting at address 1
➔ temp = 1;

temp[0] = ‘H’
temp[1] = ‘O’
temp[2] = ‘T’
temp[3] = ‘\0’

Why Pointer?

“Pointer type” of variable is an important feature pioneered by C Language:

➔To facilitate effective communication between functions (e.g., between a caller function and its sub-routine)
➔Instead of copying of a large chunk of data between functions, we only need to provide the starting address of the data.

C-String Input

- Reading in a C-string

```
const int SIZE = 10;
char Cstr[SIZE];
cin >> Cstr;     // Reads in a C-string with no
                 // blanks. Will write past the
                 // end of the array if input string
                 // is too long.
cin.getline(Cstr, 10);
                 // Reads in a C-string that may
                 // contain blanks. Ensures that <= 9
                 // chars are read in.
```

- Can also use `setw()` and `width()` to control input field widths

C-String Initialization vs. Assignment

- A C-string can be initialized at the time of its creation, just like a string object

```
const int SIZE = 10;
char month[SIZE] = "April";
```

- However, a C-string cannot later be assigned a value using the = operator; you must use the `strcpy()` function

```
char month[SIZE];
month = "August"          // wrong!
strcpy(month, "August"); //correct
```

3.11 More Mathematical Library Functions

- These require `cmath` header file
- Take `double` arguments and return a `double`
- Commonly used functions

`abs`	Absolute value
`sin`	Sine
`cos`	Cosine
`tan`	Tangent
`sqrt`	Square root
`log`	Natural (e) log

Ref:
log2(x)
log10(x)

More Mathematical Library Functions

- These require `cstdlib` header file
- `rand`
  - Returns a random number between `0` and the largest `int` the computer holds
  - Will yield the same sequence of numbers each time the program is run
- `srand(x)`
  - Initializes random number generator with `unsigned int x`
  - Should be called at most once in a program

srand(time(0));

3.12 Introduction to Files

- Can use a file instead of keyboard for program input
- Can use a file instead of monitor screen for program output
- Files are stored on secondary storage media, such as disk
- Files allow data to be retained between program executions

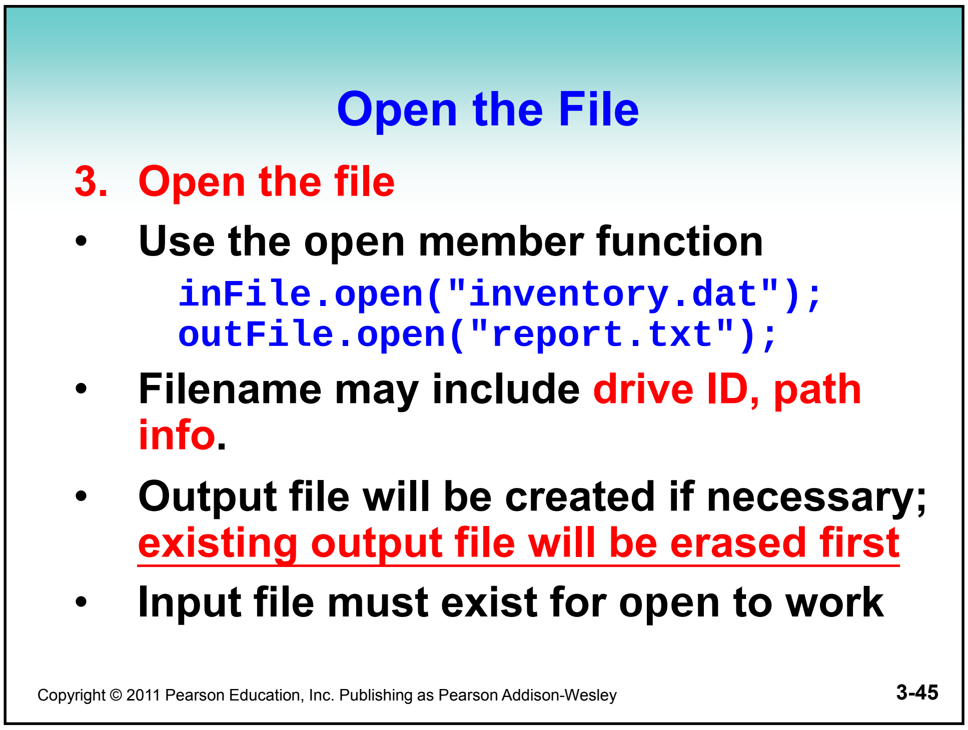
What is Needed to Use Files

1. Include the `fstream` header file
2. Define a file stream object
   - `ifstream` for input from a file

     `ifstream inFile;`
   - `ofstream` for output to a file

     `ofstream outFile;`

"inFile" and "outFile" 是
user 自己定義的 "file objects or file handles"



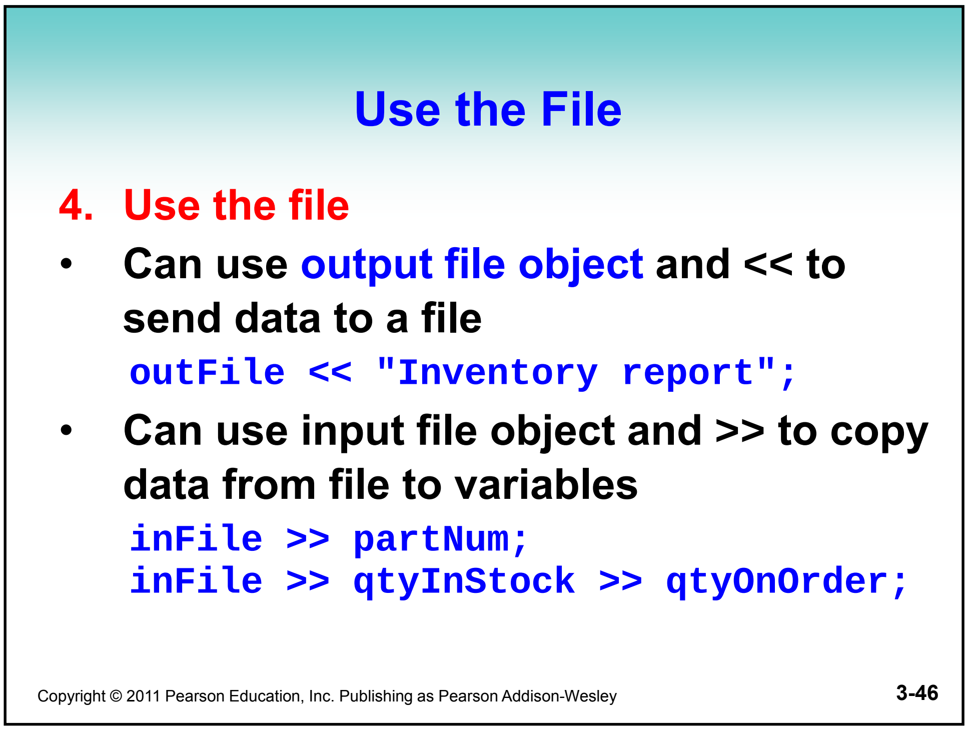
Open the File

3. Open the file

- Use the open member function

```
inFile.open("inventory.dat");
outFile.open("report.txt");
```

- Filename may include drive ID, path info.
- Output file will be created if necessary; existing output file will be erased first
- Input file must exist for open to work



Use the File

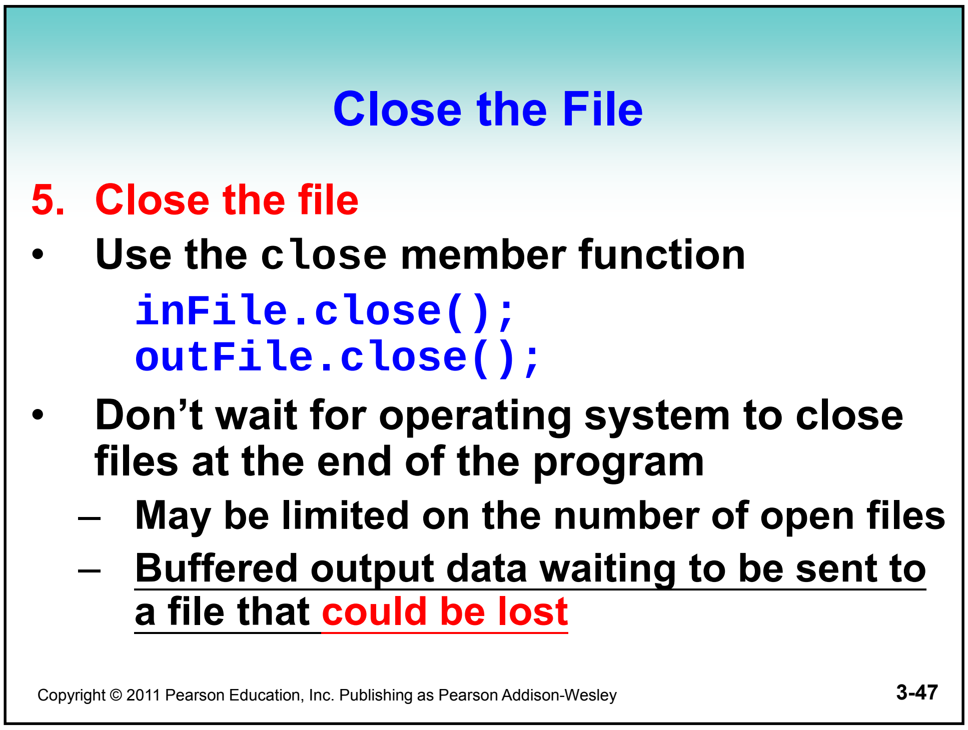
4. Use the file

- Can use output file object and << to send data to a file

```
outFile << "Inventory report";
```

- Can use input file object and >> to copy data from file to variables

```
inFile >> partNum;
inFile >> qtyInStock >> qtyOnOrder;
```



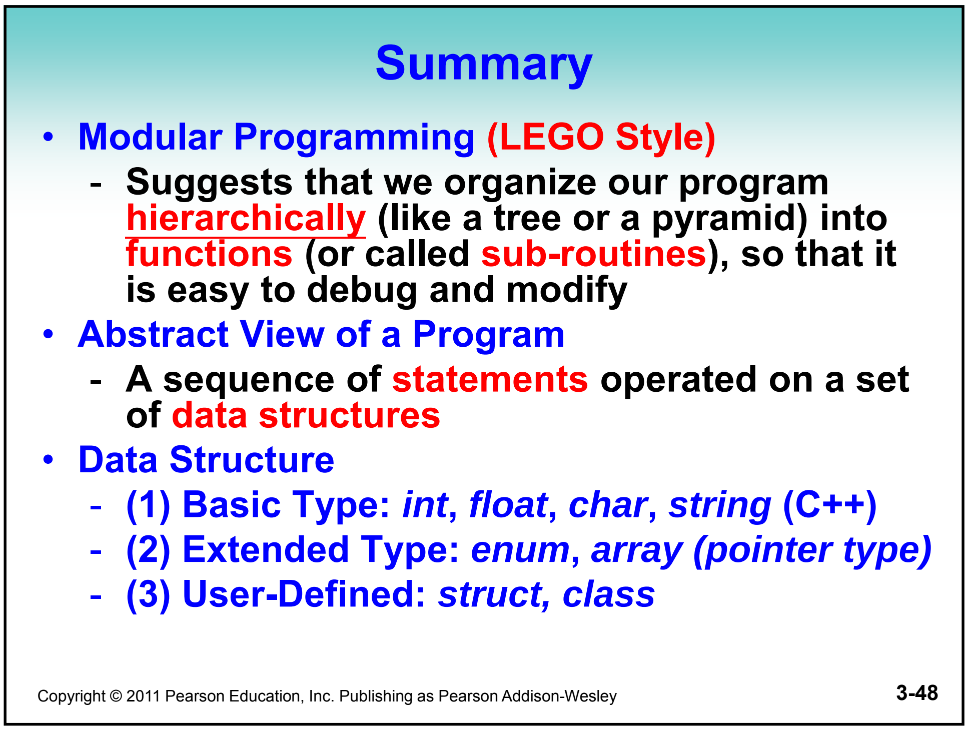
Close the File

5. Close the file

- Use the `close` member function

```
inFile.close();
outFile.close();
```

- Don't wait for operating system to close files at the end of the program
  – May be limited on the number of open files
  – Buffered output data waiting to be sent to a file that could be lost



Summary

- Modular Programming (LEGO Style)
  - Suggests that we organize our program hierarchically (like a tree or a pyramid) into functions (or called sub-routines), so that it is easy to debug and modify
- Abstract View of a Program
  - A sequence of statements operated on a set of data structures
- Data Structure
  - (1) Basic Type: *int*, *float*, *char*, *string* (C++)
  - (2) Extended Type: *enum*, *array (pointer type)*
  - (3) User-Defined: *struct, class*

Chapter 4: Making Decisions

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Addison-Wesley is an imprint of PEARSON

Topics

4.1 Relational Operators
4.2 The `if` Statement
4.3 The `if/else` Statement
4.4 The `if/else if` Statement
4.5 Menu-Driven Programs
4.6 Nested `if` Statements
4.7 Logical Operators

Topics (continued)

4.8 Validating User Input
4.9 More about Variable Definitions and Scope
4.10 Comparing Characters and Strings
4.11 The Conditional Operator
4.12 The `switch` Statement
4.13 Enumerated Data Types
4.14 Testing for File Open Errors

4.1 Relational Operators

- Used to compare numbers to determine relative order
- Operators:

`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`==`	Equal to
`!=`	Not equal to

Relational Expressions

- Relational expressions are Boolean (*i.e.*, evaluate to `true` or `false`)
- Examples:

  `12 > 5` is `true`
  `7 <= 5` is `false`

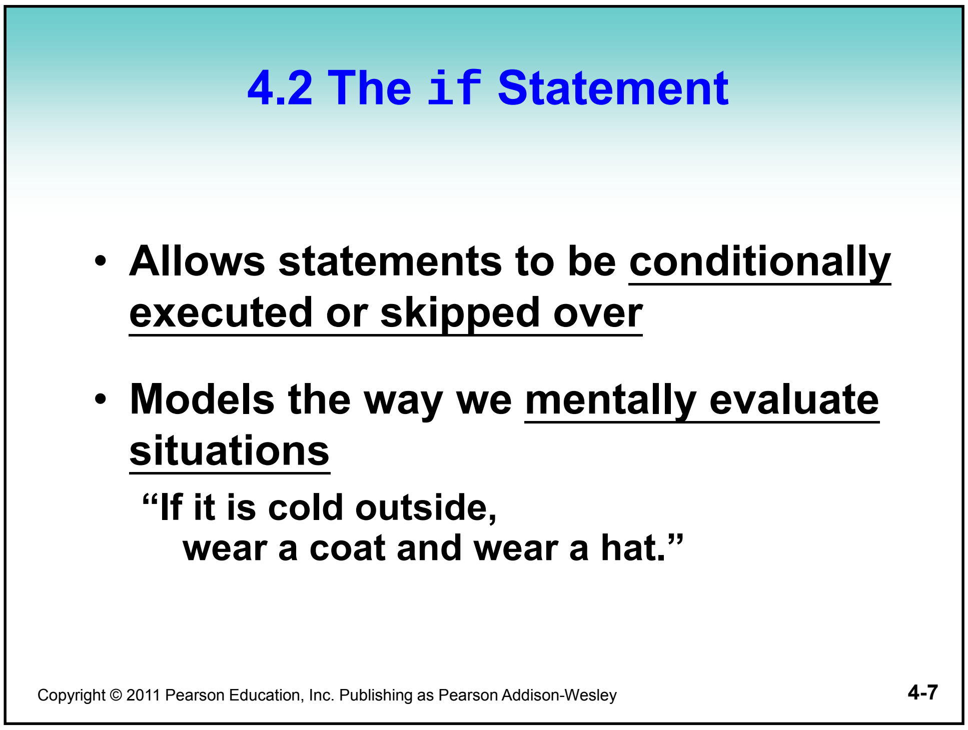
  if `x` is 10, then
  `x == 10` is `true`,
  `x != 8` is `true`, and
  `x == 8` is `false`

Relational Expressions

- Can be assigned to a variable

```
bool result = (x <= y);
```

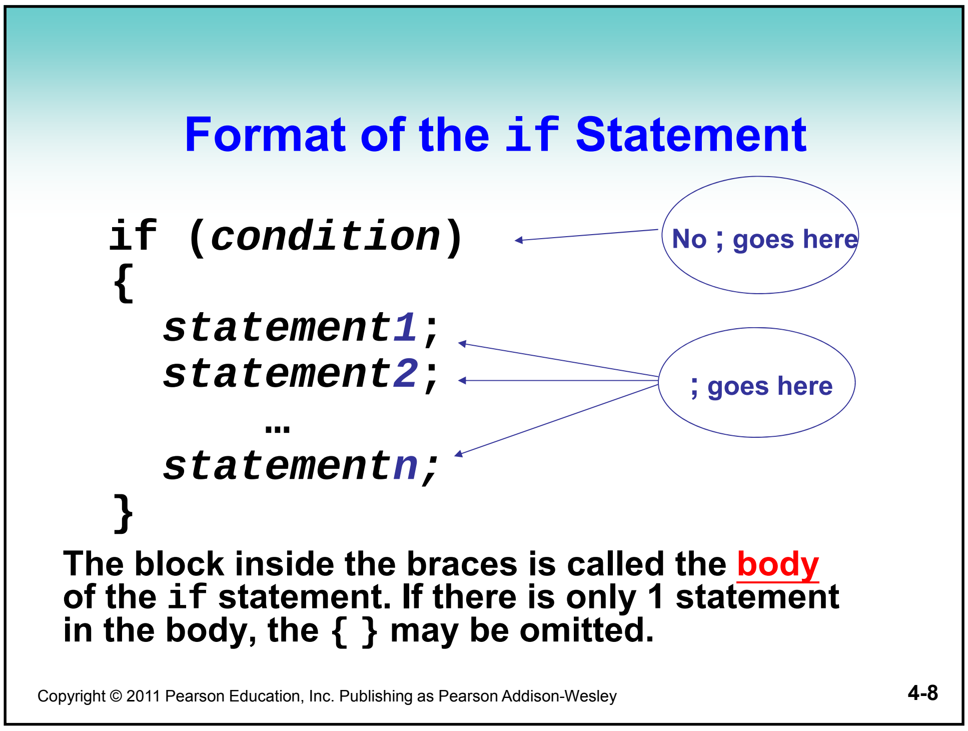
- Assigns `0` for `false`, `1` for `true`
- Do not confuse = (assignment) and == (equal to)



4.2 The `if` Statement

- Allows statements to be conditionally executed or skipped over
- Models the way we mentally evaluate situations

  "If it is cold outside,
  wear a coat and wear a hat."



Format of the `if` Statement

```
if (condition)
{
   statement1;
   statement2;
        …
   statementn;
}
```

No ; goes here

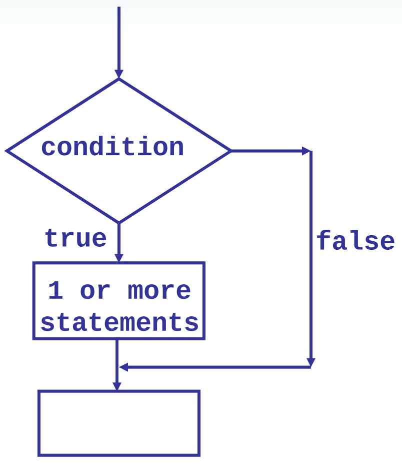
; goes here

The block inside the braces is called the body of the `if` statement. If there is only 1 statement in the body, the { } may be omitted.

How the `if` Statement Works

- If (*`condition`*) is `true`, then the *`statement(s)`* in the body are executed.
- If (*`condition`*) is `false`, then the *`statement(s)`* are skipped.

`if` Statement Flow of Control



Example `if` Statements

```
if (score >= 60)
   cout << "You passed.\n";

if (score >= 90)
{
   grade = 'A';
   cout << "Wonderful job!\n";
}
```

`if` Statement Notes

- Do not place `;` after (*`condition`*)
- Don't forget the `{ }` around a multi-statement body
- Place each *`statement;`* on a separate line after (*`condition`*), indented
- 0 is `false`; any other value is `true`

What is `true` and `false`?

- An expression whose value is 0 is considered `false`.
- An expression whose value is non-zero is considered `true`.
- An expression need not be a comparison – it can be a single variable or a mathematical expression.

Flag

- A variable that signals a condition
- Usually implemented as a `bool`
- Meaning:
  - `true`: the condition exists
  - `false`: the condition does not exist
- The flag value can be both set and tested with `if` statements

Flag Example

Example:

```
    bool validMonths = true;
           …
    if (months < 0)
       validMonths = false; // indented
           …
    if (validMonths)
       moPayment = total / months;
```

Comparisons with floating-point numbers

- It is difficult to test for equality when working with floating point numbers.
- It is better to use
  - greater than, less than tests, or
  - test to see if value is very close to a given value

4.3 The `if/else` Statement

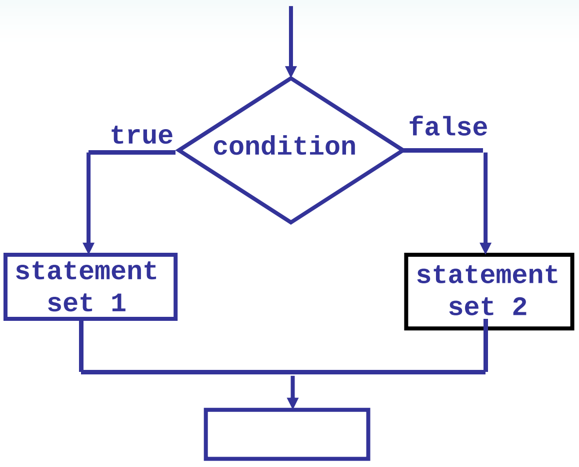
- Allows a choice between statements depending on whether (*`condition`*) is `true` or `false`
- Format:

```
if (condition)
{
   statement set 1;
}
else
{
   statement set 2;
}
```

How the `if/else` Works

- If (*`condition`*) is `true`, *`statement set 1`* is executed and *`statement set 2`* is skipped.
- If (*`condition`*) is `false`, *`statement set 1`* is skipped and *`statement set 2`* is executed.

if/else Flow of Control



Example if/else Statements

```
if (score >= 60)
   cout << "You passed.\n";
else
   cout << "You did not pass.\n";

if (intRate > 0)
{  interest = loanAmt * intRate;
   cout << interest;
}
else
   cout << "You owe no interest.\n";
```

4.4 The `if/else if` Statement

- Chain of `if` statements that test in order until one is found to be true
- Also models thought processes

  "If it is raining, take an umbrella,
  else, if it is windy, take a hat,
  else, if it is sunny, take sunglasses."

`if/else if` Format

```
if (condition 1)
{   statement set 1;
}
else if (condition 2)
{   statement set 2;
}
        …
else if (condition n)
{   statement set n;
}
```

Using a Trailing `else`

- Used with `if/else if` statement when all of the conditions are false
- Provides a default statement or action
- Can be used to catch invalid values or handle other exceptional situations

Example `if/else if` with Trailing `else`

```
if (age >= 21)
   cout << "Adult";
else if (age >= 13)
   cout << "Teen";
else if (age >= 2)
   cout << "Child";
else
   cout << "Baby";
```

4.5 Menu-Driven Program

- **Menu**: list of choices presented to the user on the computer screen
- **Menu-driven program**: program execution controlled by user selecting from a list of actions
- Menu can be implemented using `if/else if` statements

4.6 Nested `if` Statements

- An `if` statement that is part of the `if` or `else` part of another `if` statement
- Can be used to evaluate > 1 data item or condition

```
if (score < 100)
{
   if (score > 90)
      grade = 'A';
}
```

Notes on Coding Nested `if`s

- An `else` matches the nearest `if` that does not have an `else`

```
if (score < 100)
   if (score > 90)
      grade = 'A';
   else ...   // goes with second if,
              // not first one
```

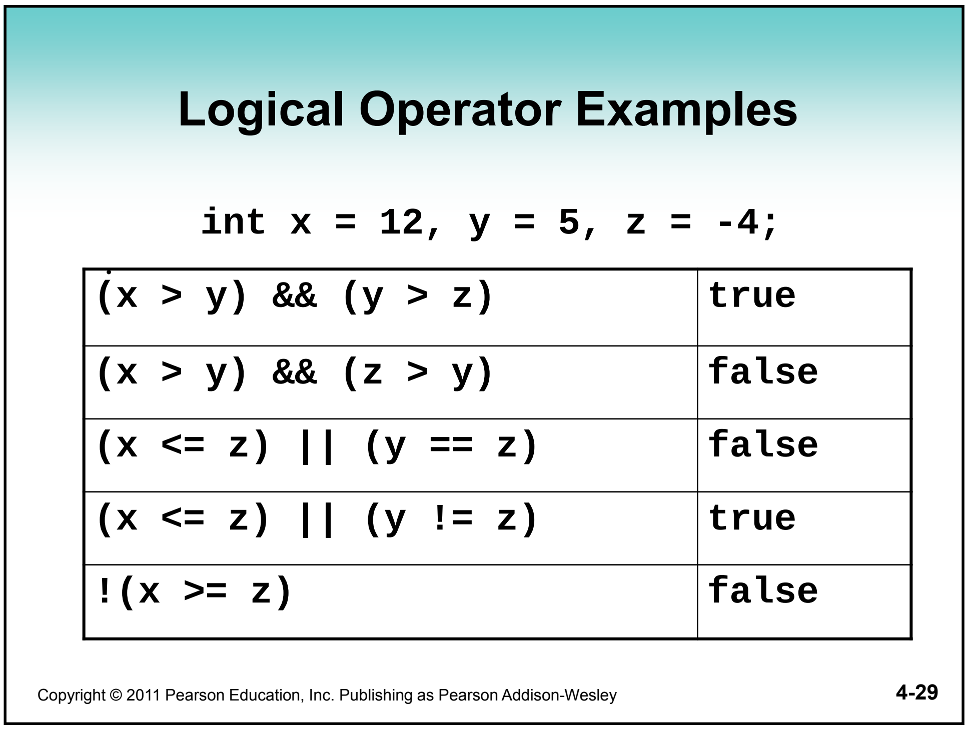
- Proper **indentation** aids comprehension
  縮排

4.7 Logical Operators

Used to create **relational expressions** from other relational expressions

Operators, Meaning, and Explanation

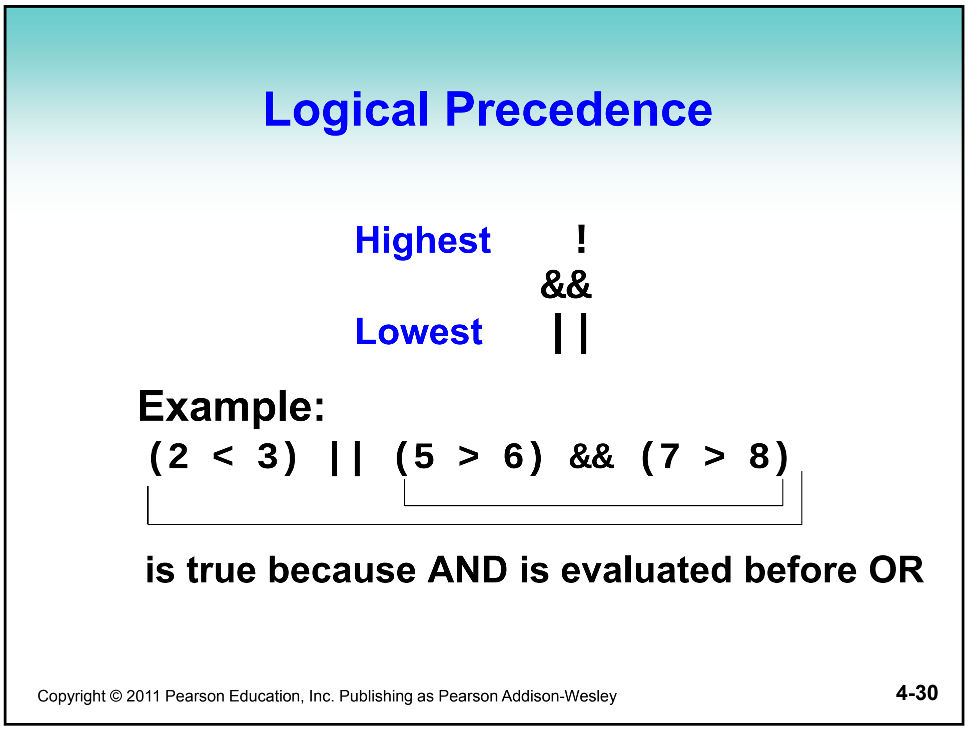
&&	AND	New relational expression is true if both expressions are true
||	OR	New relational expression is true if either expression is true
!	NOT	Reverses the value of an expression; true expression becomes false, false expression becomes true



Logical Operator Examples

```
int x = 12, y = 5, z = -4;
```

`(x > y) && (y > z)`	`true`
`(x > y) && (z > y)`	`false`
`(x <= z) || (y == z)`	`false`
`(x <= z) || (y != z)`	`true`
`!(x >= z)`	`false`



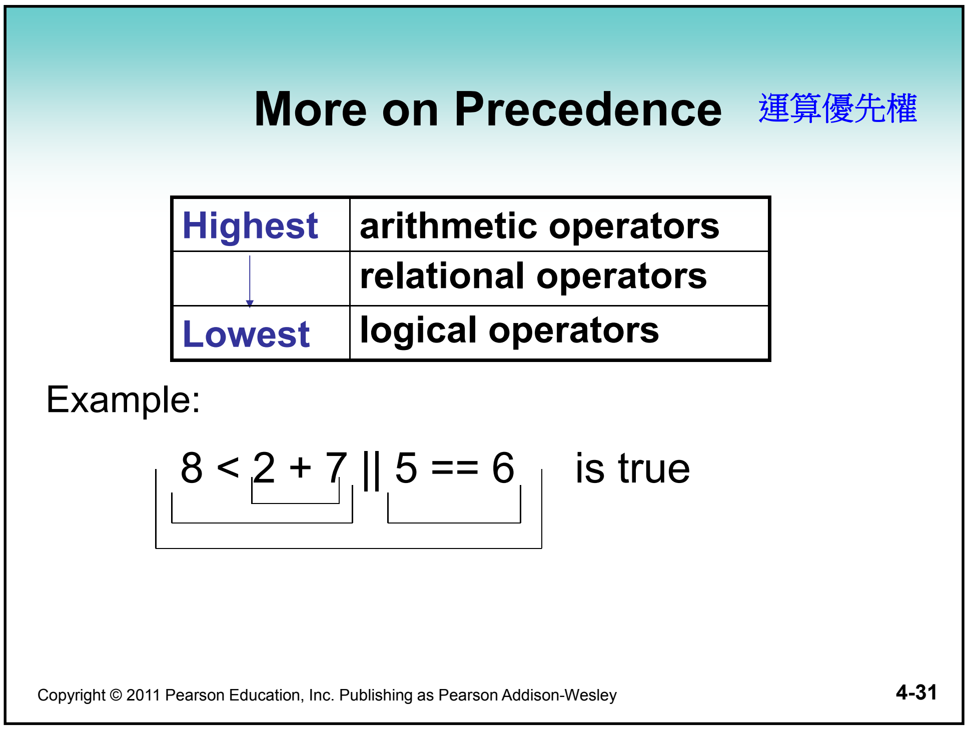
Logical Precedence

Highest	!
	&&
Lowest	||

Example:

$$\underbrace{(2 < 3) \;||\; \underbrace{(5 > 6) \;\&\&\; (7 > 8)}}$$

is true because AND is evaluated before OR

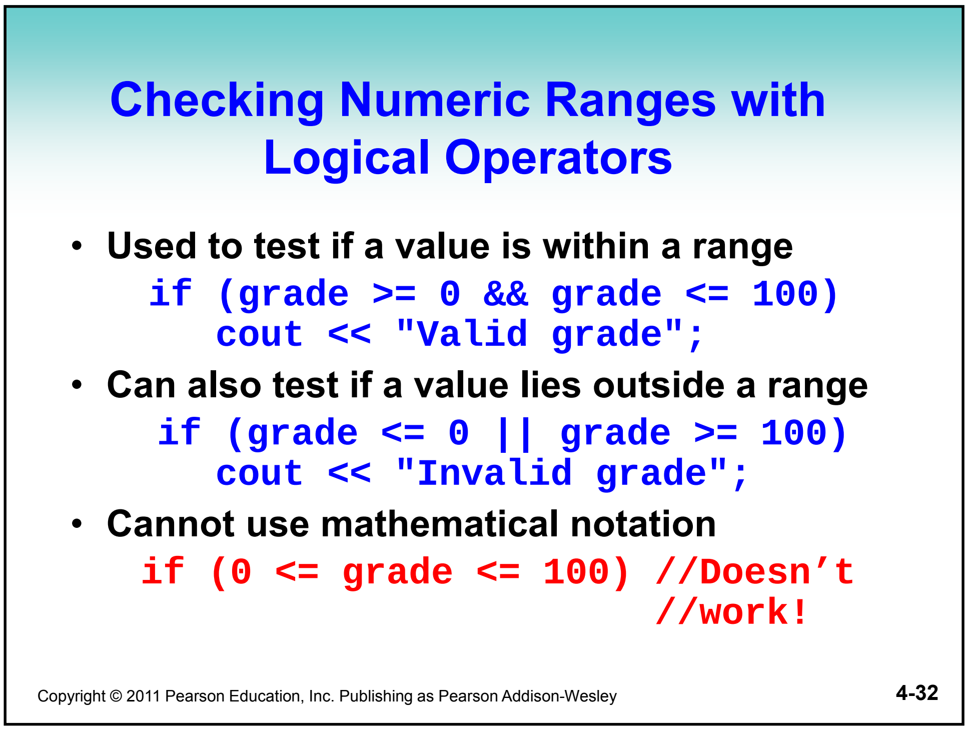


More on Precedence 運算優先權

Highest	arithmetic operators
↓	relational operators
Lowest	logical operators

Example:

$\underbrace{\underbrace{8 < \underbrace{2 + 7}} \;||\; \underbrace{5 == 6}}$ is true



Checking Numeric Ranges with Logical Operators

- Used to test if a value is within a range

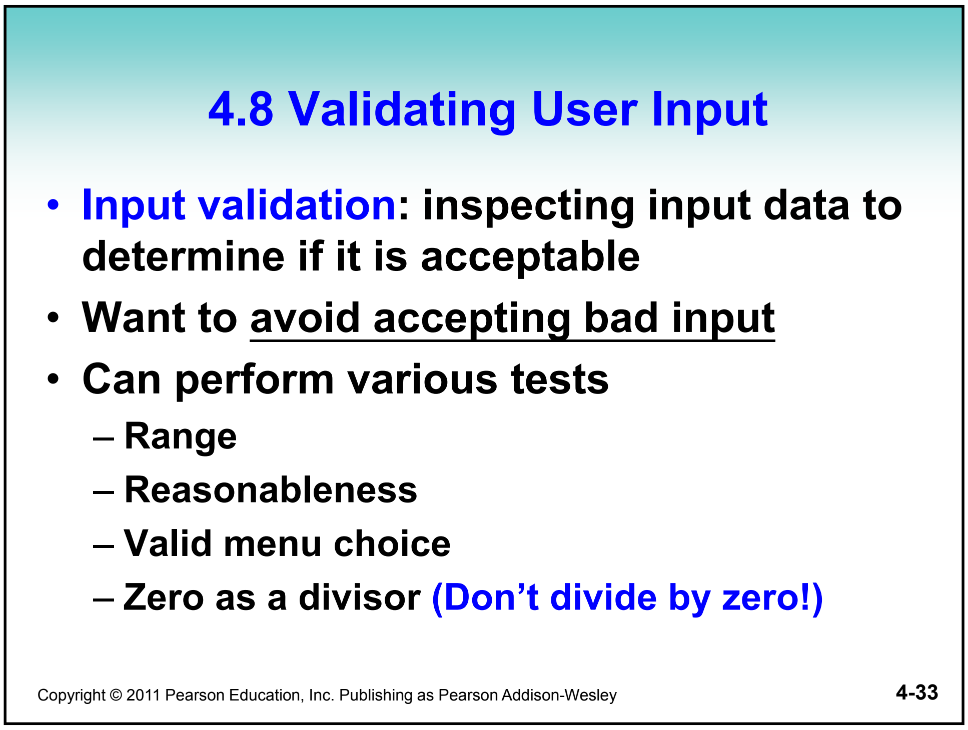
```
if (grade >= 0 && grade <= 100)
   cout << "Valid grade";
```

- Can also test if a value lies outside a range

```
if (grade <= 0 || grade >= 100)
   cout << "Invalid grade";
```

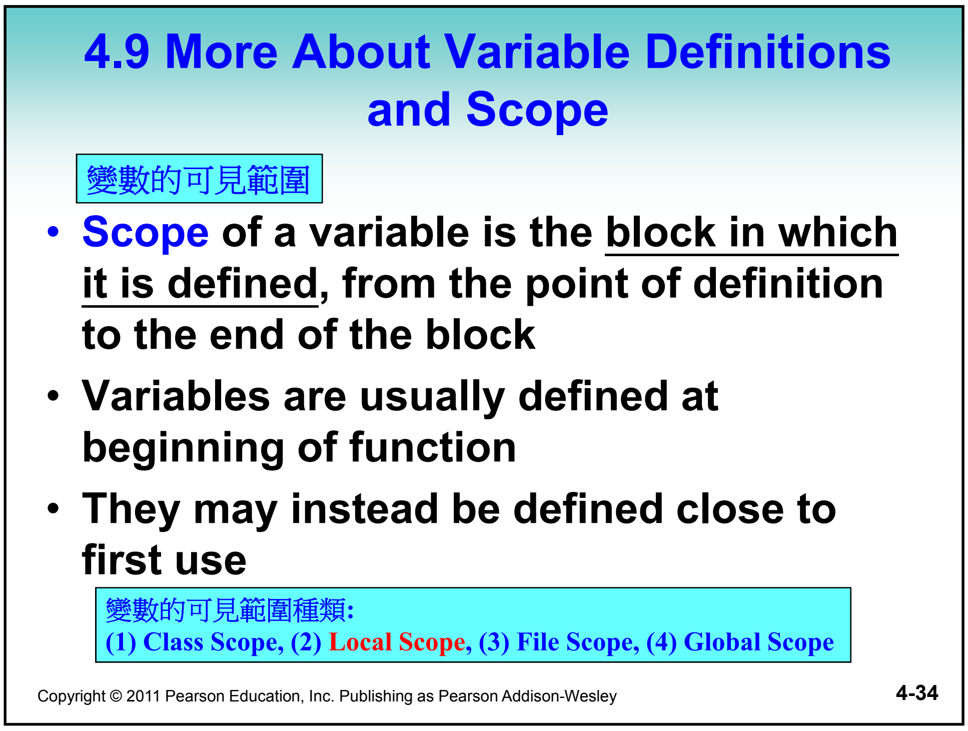
- Cannot use mathematical notation

```
if (0 <= grade <= 100) //Doesn't
                       //work!
```



4.8 Validating User Input

- Input validation: inspecting input data to determine if it is acceptable
- Want to avoid accepting bad input
- Can perform various tests
  - Range
  - Reasonableness
  - Valid menu choice
  - Zero as a divisor (Don't divide by zero!)



4.9 More About Variable Definitions and Scope

變數的可見範圍

- Scope of a variable is the block in which it is defined, from the point of definition to the end of the block
- Variables are usually defined at beginning of function
- They may instead be defined close to first use

變數的可見範圍種類:
(1) Class Scope, (2) Local Scope, (3) File Scope, (4) Global Scope

More About Variable Definitions and Scope

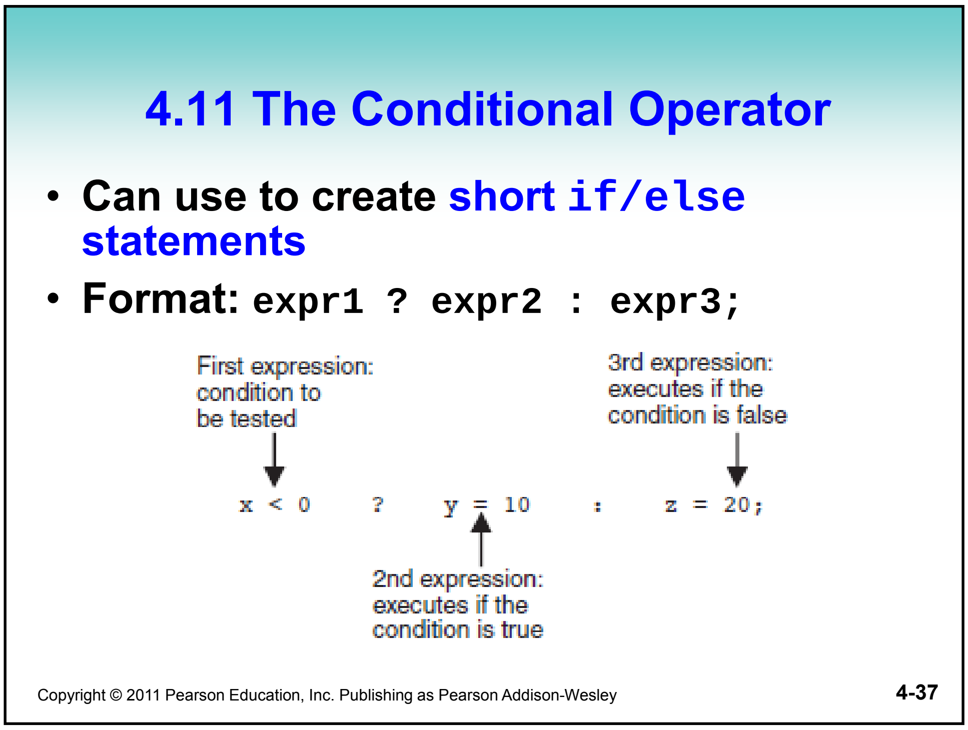
- Variables defined inside { } have local or block scope
- When in a block that is nested inside another block, you can define variables with the same name as in the outer block.
  - When the program is executing in the inner block, the outer definition is not available
  - This is generally not a good idea

4.10 Comparing Characters and Strings

- Can use relational operators with characters and string objects

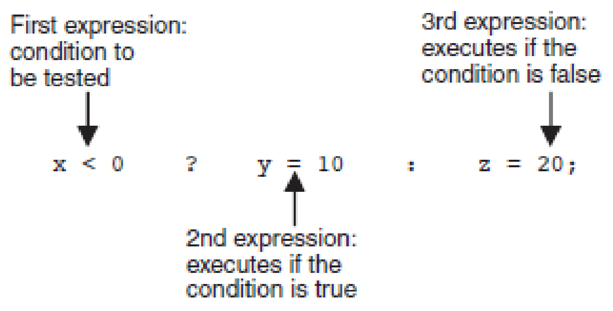
```
if (menuChoice == 'A')
if (firstName == "Beth")
```
只有C++ 可以，C 不行

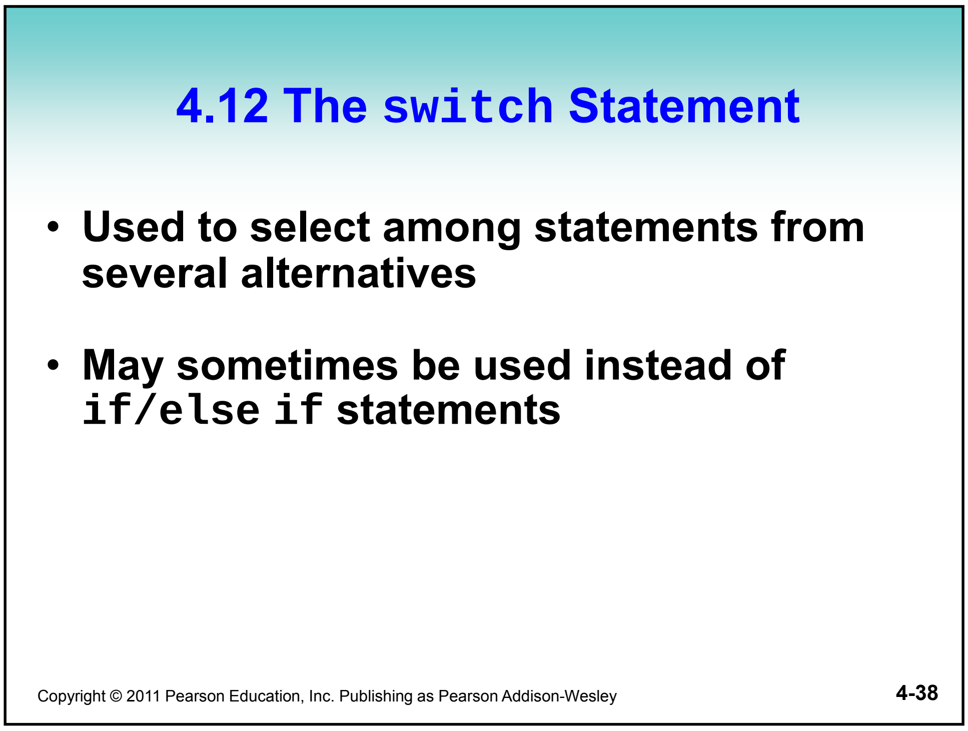
- Comparing characters is really comparing ASCII values of characters
- Comparing string objects is comparing the ASCII values of the characters in the strings. Comparison is character-by-character
- Cannot compare C-style strings with relational operators



4.11 The Conditional Operator

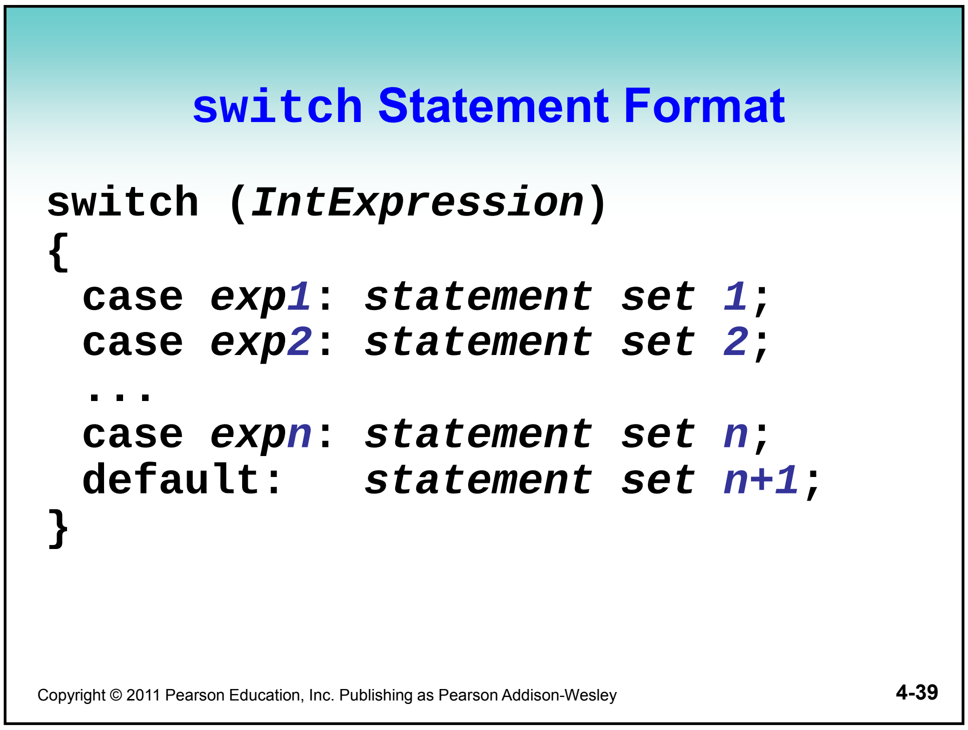
- Can use to create short `if/else` statements
- Format: `expr1 ? expr2 : expr3;`


First expression: condition to be tested
3rd expression: executes if the condition is false
x < 0 ? y = 10 : z = 20;
2nd expression: executes if the condition is true



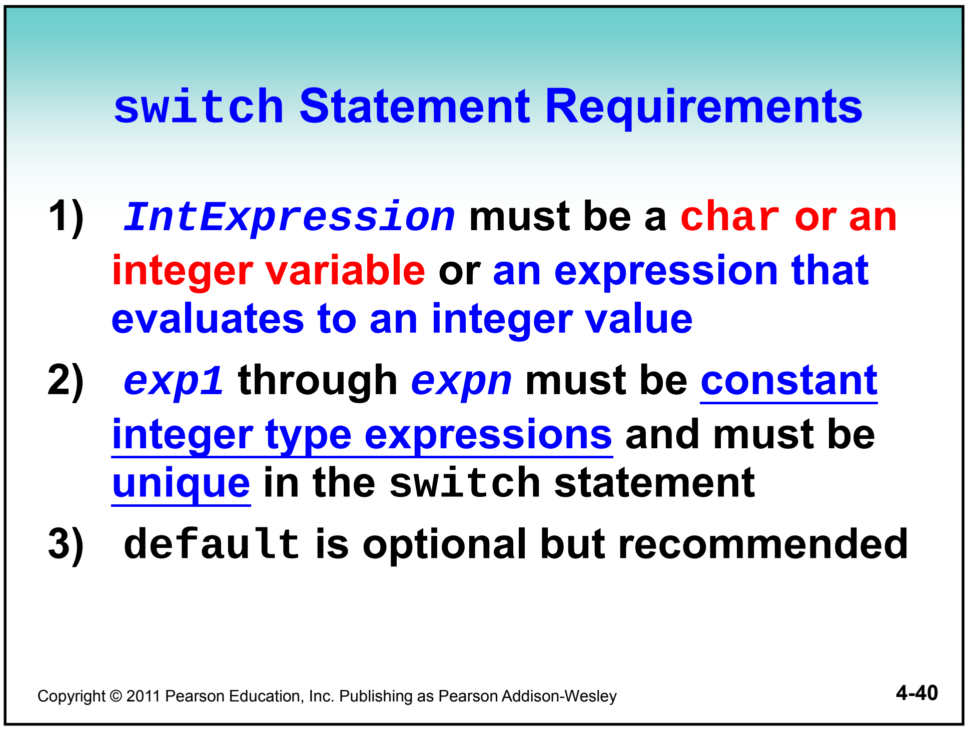
4.12 The `switch` Statement

- Used to select among statements from several alternatives
- May sometimes be used instead of `if/else if` statements



switch Statement Format

```
switch (IntExpression)
{
  case exp1: statement set 1;
  case exp2: statement set 2;
  ...
  case expn: statement set n;
  default:   statement set n+1;
}
```



switch Statement Requirements

1) ***IntExpression*** **must be a char or an integer variable or an expression that evaluates to an integer value**
2) ***exp1*** **through *expn* must be constant integer type expressions and must be unique in the switch statement**
3) **default is optional but recommended**

How the `switch` Statement Works

1) ***`IntExpression`*** is evaluated
2) The value of ***`intExpression`*** is compared against ***`exp1`*** through ***`expn`***.
3) If ***`IntExpression`*** matches value ***`expi`***, the program branches to the statement(s) following ***`expi`*** and continues to the end of the `switch`
4) If no matching value is found, the program branches to the statement after `default`:

The `break` Statement

- Used to stop execution in the current block
- Also used to exit a `switch` statement
- Useful to execute a single case statement without executing statements following it

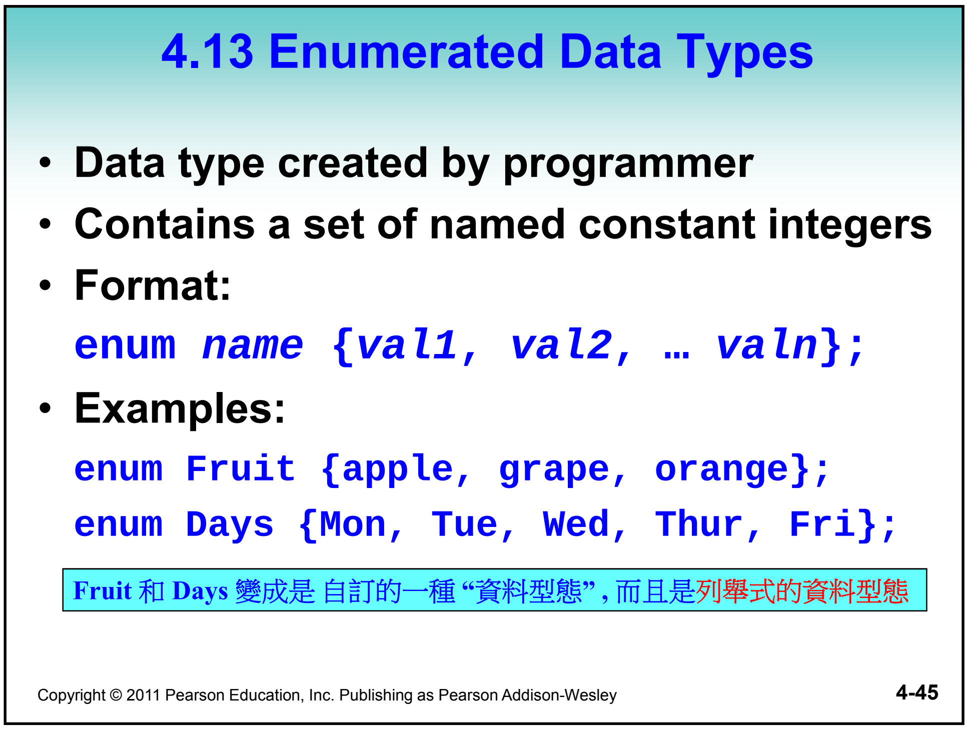
Example switch Statement

```
switch (gender)
{
  case 'f': cout << "female";
            break;
  case 'm': cout << "male";
            break;
  default : cout << "invalid gender";
}
```

Using switch with a Menu

`switch` statement is a natural choice for menu-driven program

- display menu
- get user input
- use user input as `IntExpression` in `switch` statement
- use menu choices as exp to test against in the case statements



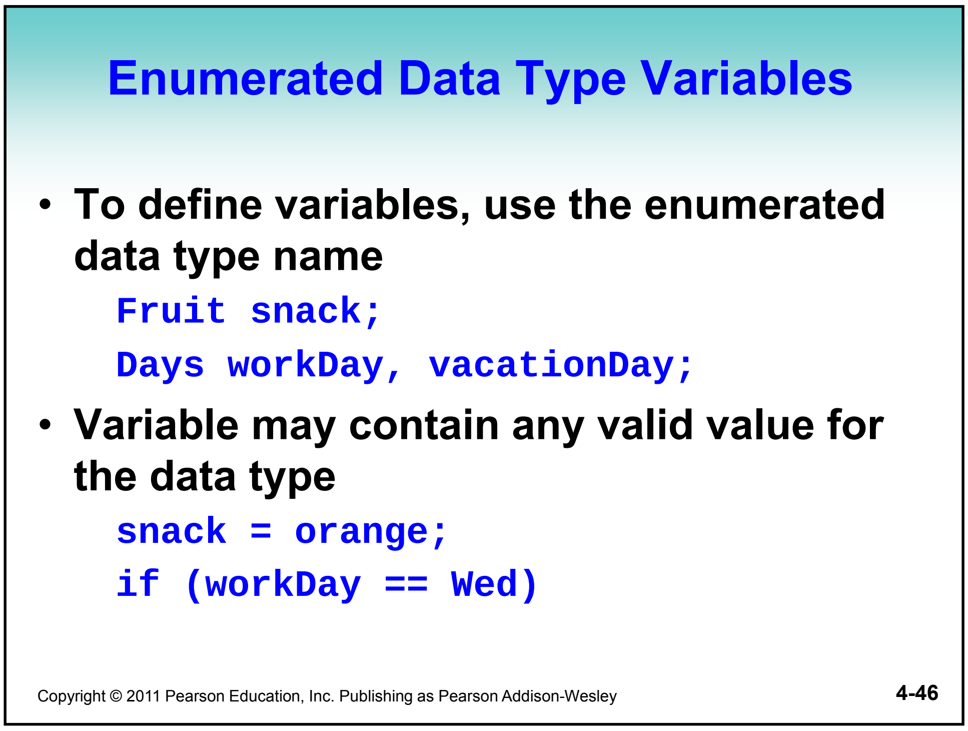
4.13 Enumerated Data Types

- Data type created by programmer
- Contains a set of named constant integers
- Format:

  `enum` *`name`* `{`*`val1, val2, … valn`*`};`
- Examples:

```
enum Fruit {apple, grape, orange};
enum Days {Mon, Tue, Wed, Thur, Fri};
```

Fruit 和 Days 變成是 自訂的一種 "資料型態", 而且是列舉式的資料型態



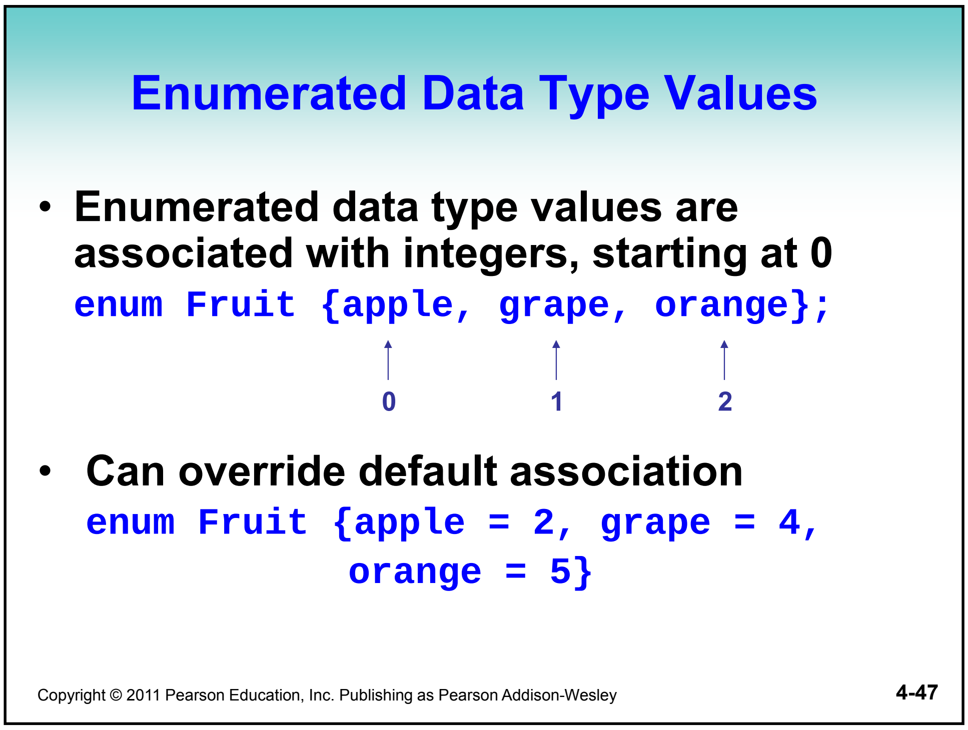
Enumerated Data Type Variables

- To define variables, use the enumerated data type name

```
Fruit snack;
Days workDay, vacationDay;
```

- Variable may contain any valid value for the data type

```
snack = orange;
if (workDay == Wed)
```



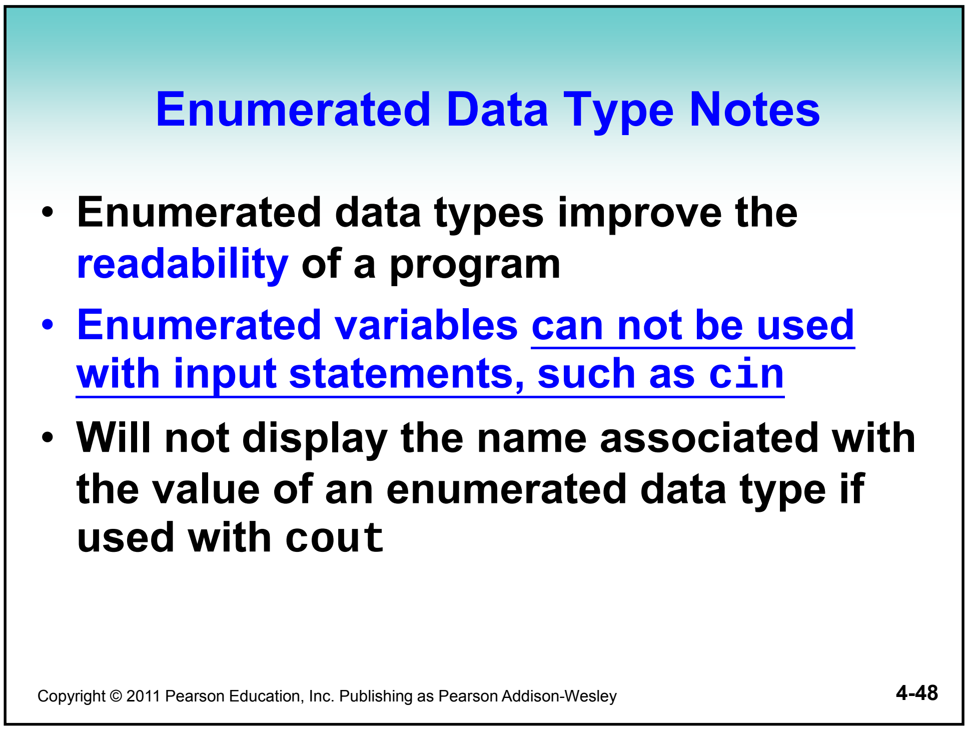
Enumerated Data Type Values

- Enumerated data type values are associated with integers, starting at 0

```
enum Fruit {apple, grape, orange};
              ↑      ↑       ↑
              0      1       2
```

- Can override default association

```
enum Fruit {apple = 2, grape = 4,
                 orange = 5}
```



Enumerated Data Type Notes

- Enumerated data types improve the readability of a program
- Enumerated variables can not be used with input statements, such as `cin`
- Will not display the name associated with the value of an enumerated data type if used with `cout`

4.14 Testing for File Open Errors

After opening a file, test that it was actually found and opened before trying to use it

- By testing the file stream object
- By using the `fail()` function

Testing the File Stream Object

Example:

```
ifstream datafile;
datafile.open("customer.dat");
if (!datafile)
   cout << "Error opening file.\n";
else
   // proceed to use the file
```

Using the `fail()` Function

Example:

```
ifstream datafile;
datafile.open("customer.dat");
if (datafile.fail())
   cout << "Error opening file.\n";
else
   // proceed to use the file
```

Chapter 5: Looping

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Topics

5.1 The Increment and Decrement Operators
5.2 Introduction to Loops: The `while` Loop
5.3 Using the `while` loop for Input Validation
5.4 Counters
5.5 The `do-while` loop
5.6 The `for` loop
5.7 Keeping a Running Total

Topics (continued)

5.8 Sentinels 哨兵
5.9 Using a Loop to Read Data From a File
5.10 Deciding Which Loop to Use
5.11 Nested Loops
5.12 Breaking Out of a Loop
5.13 The `continue` Statement
5.14 Creating Good Test Data

5.1 The Increment and Decrement Operators

- ++ adds one to a variable

  `val++;` is the same as `val = val + 1;`
- -- subtracts one from a variable

  `val--;` is the same as `val = val – 1;`
- can be used in **prefix mode** (put before a variable) or **postfix mode** (put after a variable)

Prefix Mode

- `++val` and `--val` increment or decrement the variable, *then* return the new value of the variable.
- It is this returned new value of the variable that is used in any other operations within the same statement

Prefix Mode Example

```
int x = 1, y = 1;

x = ++y;         // y is incremented to 2
                 // Then 2 is assigned to x
cout << x
  << "  " << y; // Displays 2  2

x = --y;         // y is decremented to 1
                 // Then 1 is assigned to x
cout << x
  << "  " << y; // Displays 1 1
```

Postfix Mode

- `val++` and `val--` return the old value of the variable, *then* increment or decrement the variable
- It is this returned old value of the variable that is used in any other operations within the same statement

Postfix Mode Example

```
int x = 1, y = 1;

x = y++;        // y++ returns a 1
                // The 1 is assigned to x
                // and y is incremented to 2
cout << x
  << "  " << y; // Displays 1  2

x = y--;        // y-- returns a 2
                // The 2 is assigned to x
                // and y is decremented to 1
cout << x
  << "  " << y; // Displays 2 1
```

Increment & Decrement Notes

- Can be used in arithmetic expressions
  `result = num1++ + --num2;`
- Must be applied to something that has a location in memory. Cannot have
  `result = (num1 + num2)++; // Illegal`
- Can be used in relational expressions
  `if (++num > limit)`
- Pre- and post-operations will cause different comparisons

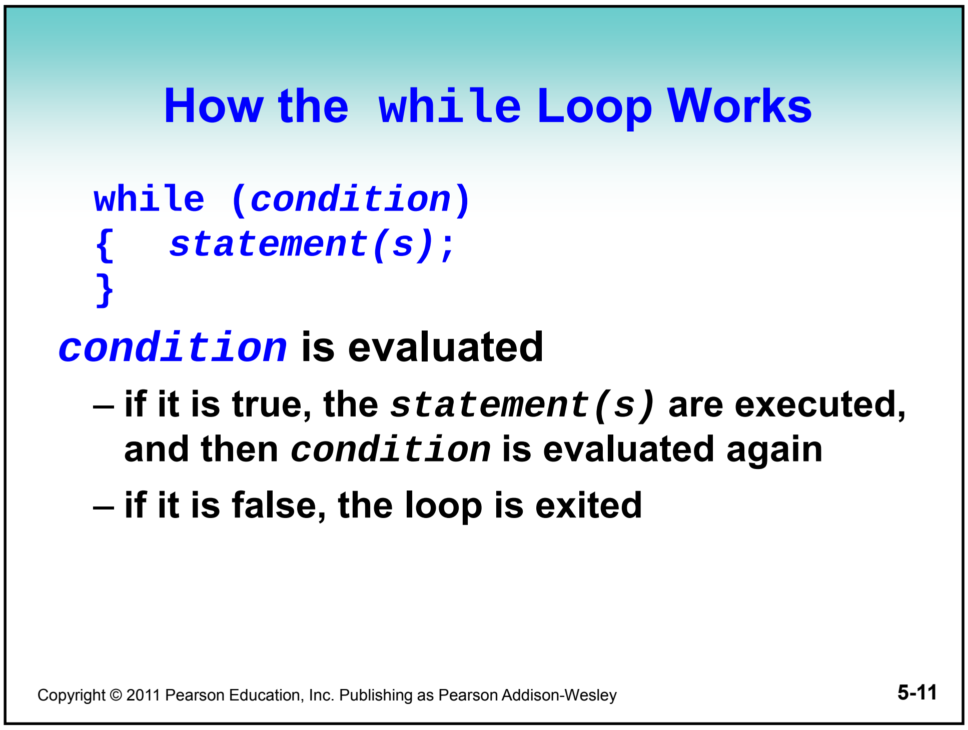
5.2 Introduction to Loops: The `while` Loop

- Loop: part of program that may execute > 1 time (*i.e.*, it repeats)
- `while` loop format:

```
while (condition)
{  statement(s);
}
```

No ';' here

- The {} can be omitted if there is only one statement in the body of the loop

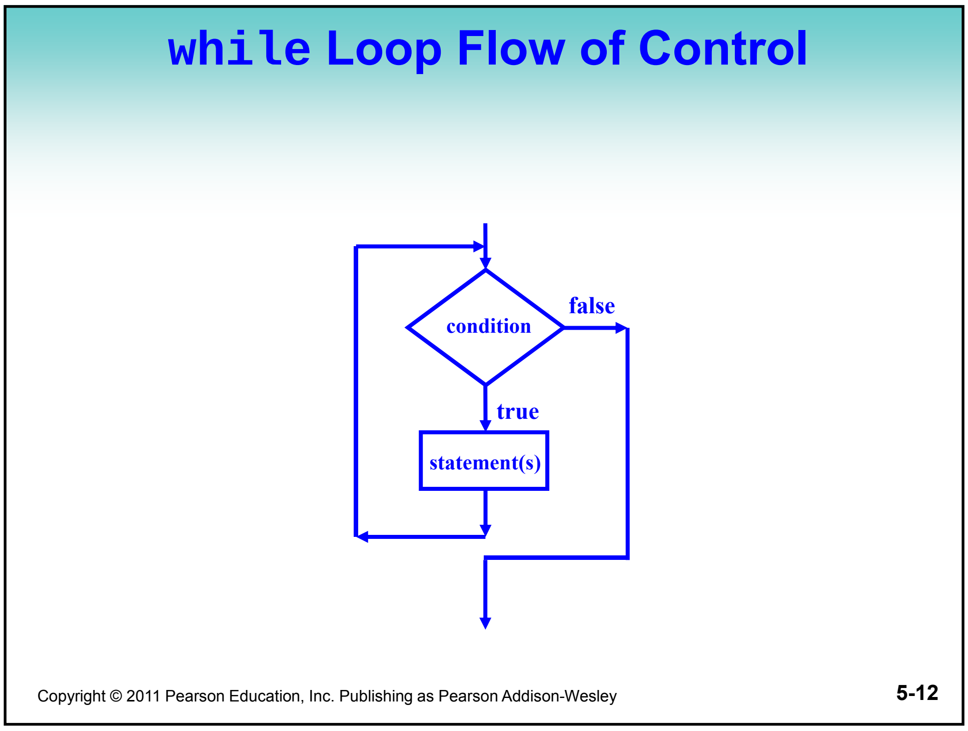


How the `while` Loop Works

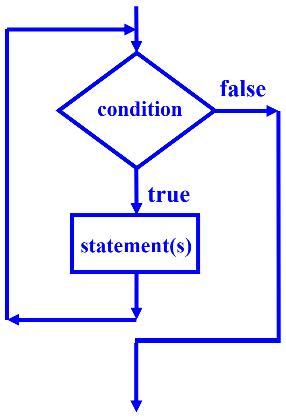
```
while (condition)
{   statement(s);
}
```

`condition` is evaluated

- if it is true, the *`statement(s)`* are executed, and then *`condition`* is evaluated again
- if it is false, the loop is exited



`while` Loop Flow of Control



while Loop Example

```
int val = 5;
while (val >= 0)
{   cout << val << "  ";
    val--;
}
```

- produces output:

```
 5  4  3  2  1  0
```

while Loop is a Pretest Loop

- `while` is a pretest loop (*`condition`* is evaluated <u>before</u> the loop executes)
- If the condition is initially false, the statement(s) in the body of the loop are never executed
- If the condition is initially true, the statement(s) in the body continue to be executed until the condition becomes false

Exiting the Loop

- The loop must contain code to allow *`condition`* to eventually become `false` so the loop can be exited
- Otherwise, you have an infinite loop (*i.e.*, a loop that does not stop)
- Example of infinite loop: 常發生的錯誤

```
x = 5;
while (x > 0)      // infinite loop because
   cout << x;      // x is always > 0
```

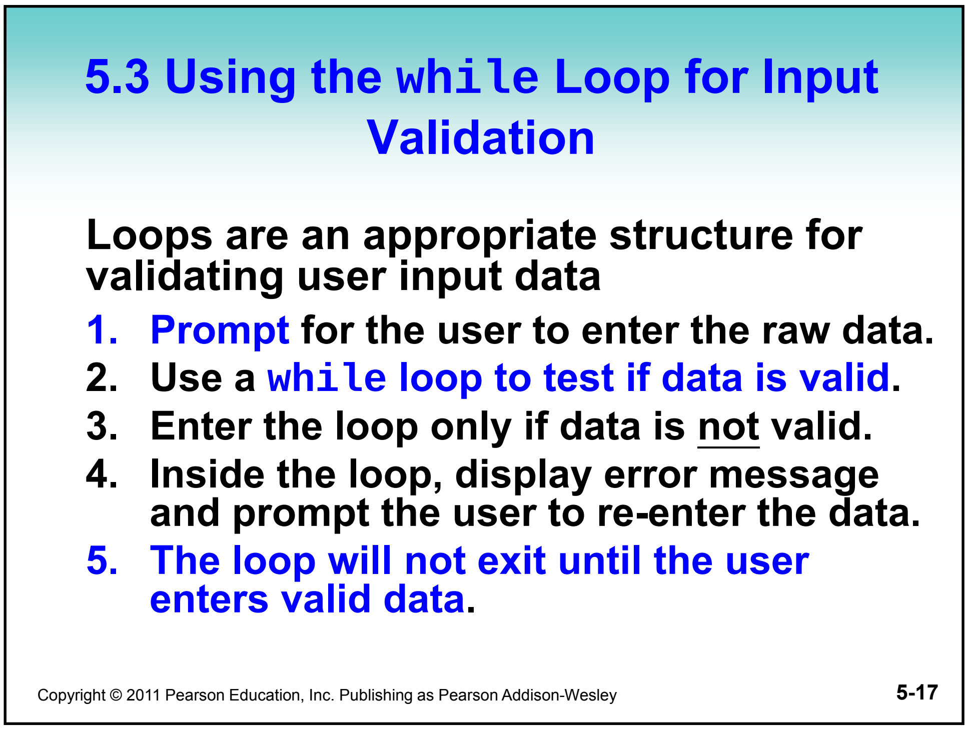
Common Loop Errors

- Don't forget the { } :

```
int numEntries = 1;
while (numEntries <=3)
   cout << "Still working … ";
   numEntries++; // not in the loop body
```

- Don't use = when you mean to use ==

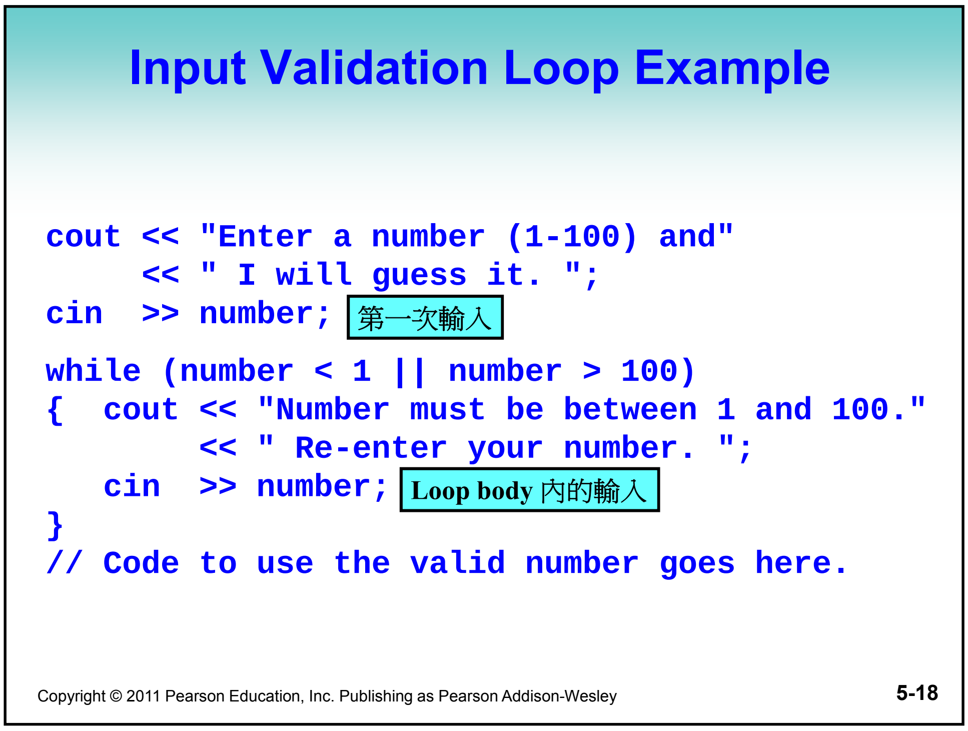
```
while (numEntries = 3)  // always true
{
   cout << "Still working … ";
   numEntries++;
}
```



5.3 Using the `while` Loop for Input Validation

Loops are an appropriate structure for validating user input data

1. Prompt for the user to enter the raw data.
2. Use a `while` loop to test if data is valid.
3. Enter the loop only if data is not valid.
4. Inside the loop, display error message and prompt the user to re-enter the data.
5. The loop will not exit until the user enters valid data.



Input Validation Loop Example

```
cout << "Enter a number (1-100) and"
     << " I will guess it. ";
cin  >> number; 第一次輸入
while (number < 1 || number > 100)
{  cout << "Number must be between 1 and 100."
        << " Re-enter your number. ";
   cin  >> number; Loop body 內的輸入
}
// Code to use the valid number goes here.
```

5.4 Counters

控制迴圈的執行次數

- **Counter**: variable that is incremented or decremented each time a loop is executed
- Can be used to control execution of the loop (**loop control variable**)
- **Must be initialized** before entering the loop
- May be incremented/decremented either inside the loop or in the loop test

User Controls the Loop Example

```
int num, limit;
cout << "Table of squares\n";
cout << "How high to go? ";
cin  >> limit; //user control this
cout << "\n\nnumber square\n";
num = 1; 起始 counter
while (num <= limit)
{  cout << setw(5) << num << setw(6)
        << num*num << endl;
   num++; 更新 counter
}
```

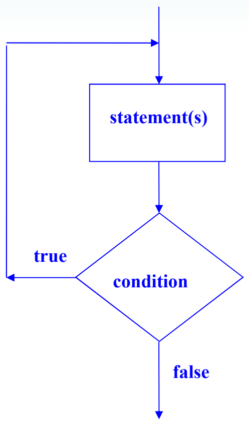
5.5 The do-while Loop

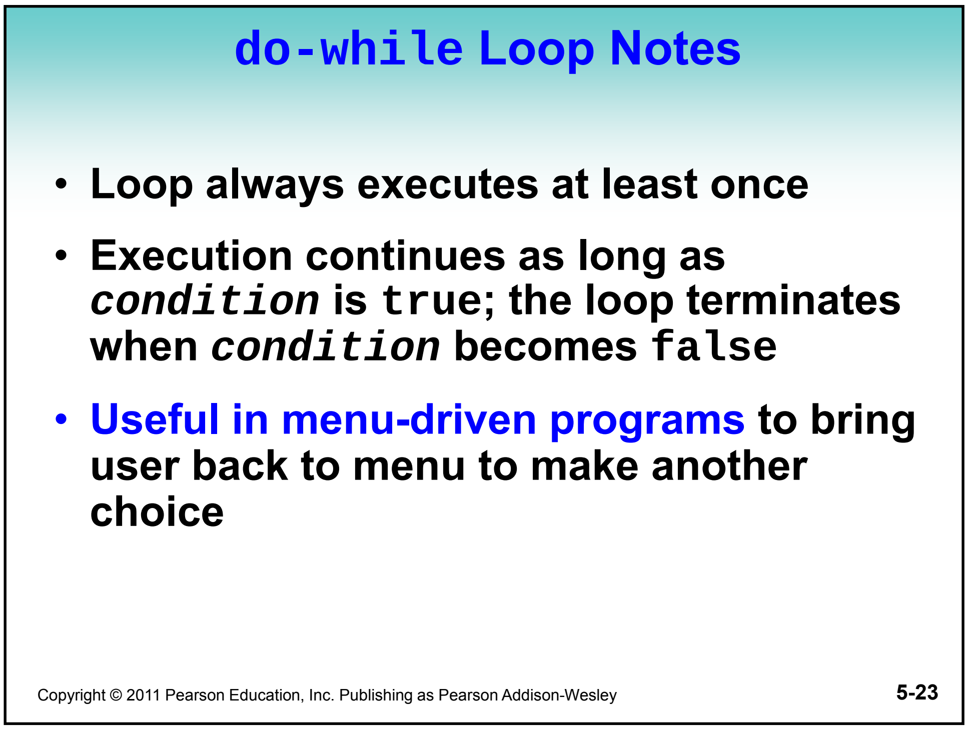
- do-while: a post-test loop 至少執行一次
  (*condition* is evaluated after the loop is executed)
- Format:

```
do
{   1 or more statements;
} while (condition);
```

Notice the required ;

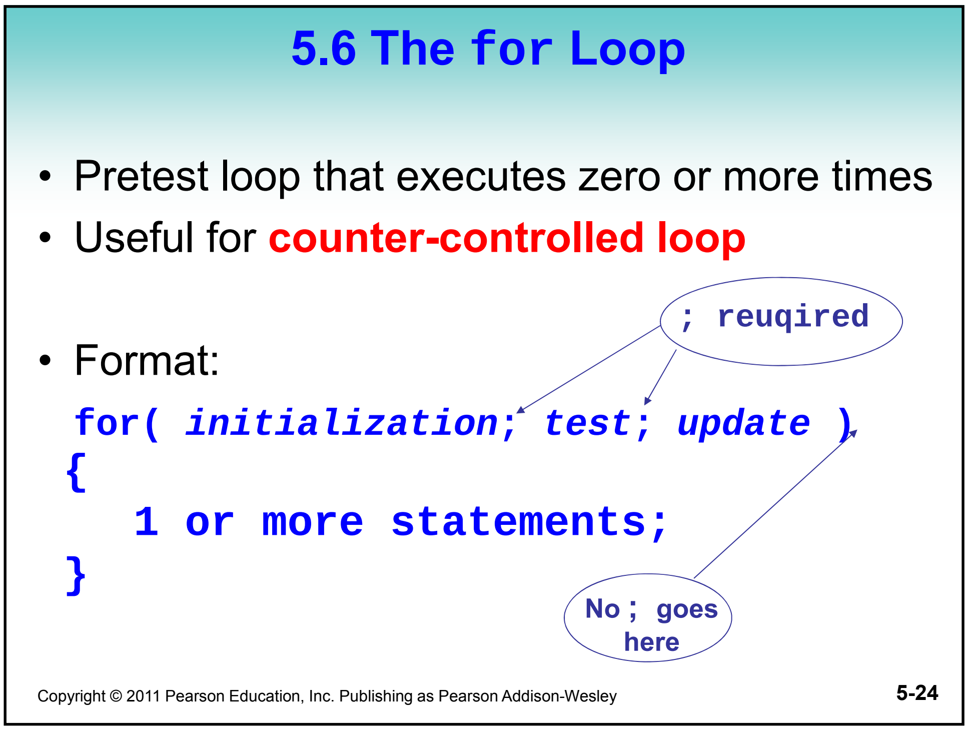
do-while Flow of Control





`do-while` Loop Notes

- Loop always executes at least once
- Execution continues as long as *`condition`* is `true`; the loop terminates when *`condition`* becomes `false`
- **Useful in menu-driven programs** to bring user back to menu to make another choice



5.6 The `for` Loop

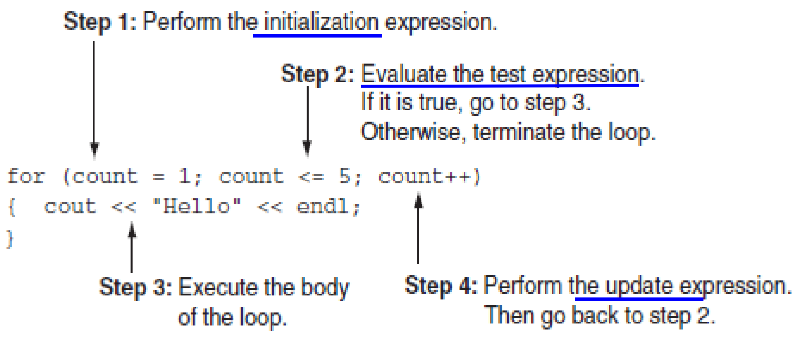
- Pretest loop that executes zero or more times
- Useful for **counter-controlled loop**
- Format:

```
for( initialization; test; update )
{
   1 or more statements;
}
```

`; reuqired`

No ; goes here

for Loop Mechanics



Step 1: Perform the initialization expression.

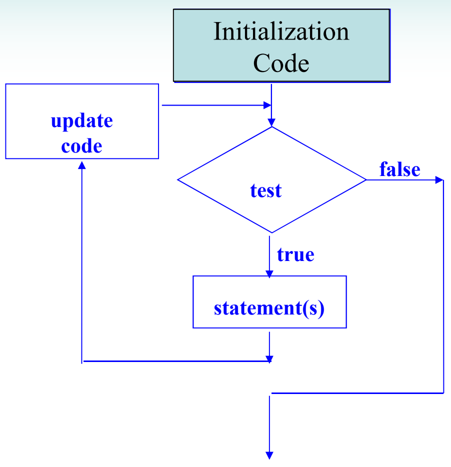
Step 2: Evaluate the test expression.
If it is true, go to step 3.
Otherwise, terminate the loop.

```
for (count = 1; count <= 5; count++)
{  cout << "Hello" << endl;
}
```

Step 3: Execute the body of the loop.

Step 4: Perform the update expression. Then go back to step 2.

for Loop Flow of Control



for Loop Example

```
int sum = 0, num;
for (num = 1; num <= 10; num++)
    sum += num;
cout << "Sum of numbers 1 – 10 is "
     << sum << endl;
```

for Loop Notes

- If *test* is false the first time it is evaluated, the body of the loop will not be executed at all
- The update expression can increment or decrement by any amount

`for` Loop Modifications

- Can define variables in initialization code
  - Their scope is the `for` loop
- Initialization and update code can contain more than one statement
  - Separate statements with commas
- Example:

```
for (int sum = 0, num = 1; num <= 10; num++)
    sum += num;
```

More `for` Loop Modifications
(These are NOT Recommended)

- Can omit ***`initialization`*** if already done

```
int sum = 0, num = 1;
for (; num <= 10; num++)
    sum += num;
```

- Can omit ***`update`*** if done in loop

```
for (sum = 0, num = 1; num <= 10;)
    sum += num++;
```

- Can omit ***`test`*** – may cause an infinite loop

```
for (sum = 0, num = 1; ; num++)
    sum += num;
```

- Can omit loop body if all work is done in the header

5.7 Keeping a Running Total

- **running total**: accumulated sum of numbers from each repetition of loop
- **accumulator**: variable that holds running total

```
int sum = 0, num = 1; // sum is the
while (num <= 10)     // accumulator
{   sum += num;
    num++;
}
cout << "Sum of numbers 1 – 10 is "
     << sum << endl;
```

5.8 Sentinels

- **Sentinel**: value in a list of values that indicates end of data
- Special value that cannot be confused with a valid value, *e.g.*, -999 for a test score
- Used to terminate the input process

Sentinel Example

```
int total = 0;
cout << "Enter points earned "
     << "(or -1 to quit): ";
cin  >> points;
while (points != -1) // -1 is the sentinel
{
   total += points;
   cout << "Enter points earned: ";
   cin  >> points;
}
```

5.9 Using a Loop to Read Data From a File

- A Loop can be used to read in each piece of data from a file
- It is not necessary to know how much data is in the file
- Several methods exist to test for the end of the file

Using the `eof()` Function to Test for the End of a File

- **`eof()` member function returns `true` when the previous read encountered the end of file; returns `false` otherwise**
- **Example:**

```
datafile >> score;
while (!datafile.eof())
{
   sum += score;
   datafile >> score;
}
```

Problems Using `eof()`

- **For the `eof()` function to work correctly using this method, there must be a whitespace (space, tab, or [Enter]) after the last piece of data**
- **Otherwise the end of file will be encountered when reading the final data value and it will not be processed**

Using the >> Operation

- The stream extraction operator (>>) returns a value indicating if a read is successful
- This can be tested to find the end of file since the read "fails" when there is no more data
- Example:

```
while (datafile >> score)
   sum += score;
```

5.10 Deciding Which Loop to Use

- `while`: pretest loop (loop body may not be executed at all)
- `do-while`: post test loop (loop body will always be executed at least once)
- `for`: pretest loop (loop body may not be executed at all); has initialization and update code; is useful with counters or if precise number of repetitions is known

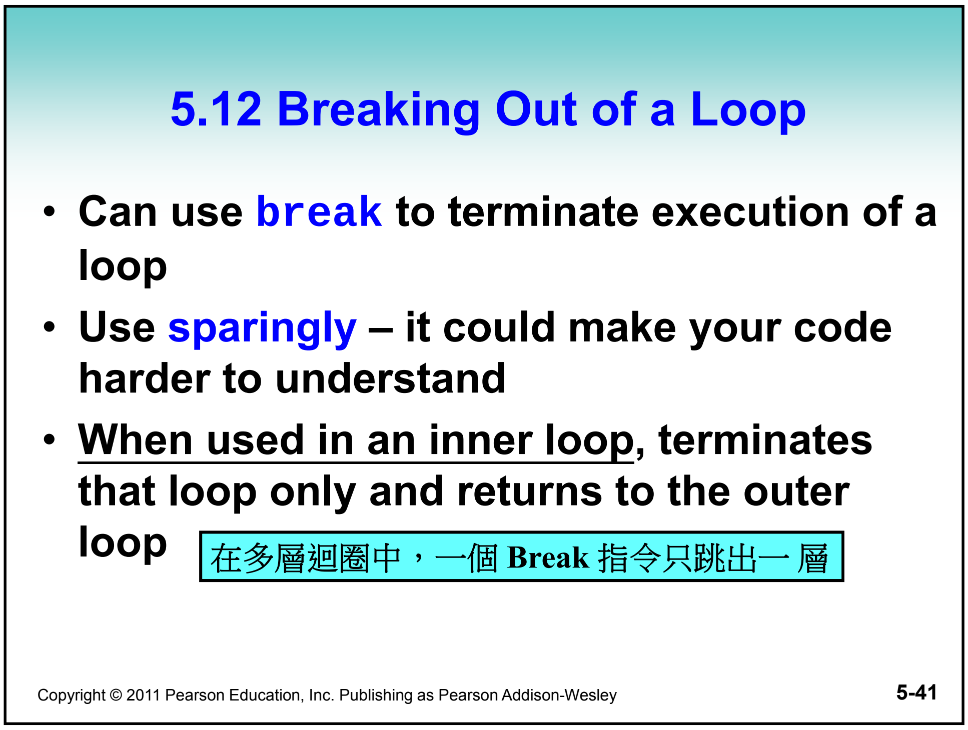
5.11 Nested Loops

- A nested loop is a loop inside the body of another loop
- Example:

```
for (row = 1; row <= 3; row++)        outer loop
{
  for (col = 1; col <= 3; col++)      inner loop
  {
     cout << row * col << endl;
  }
}
```

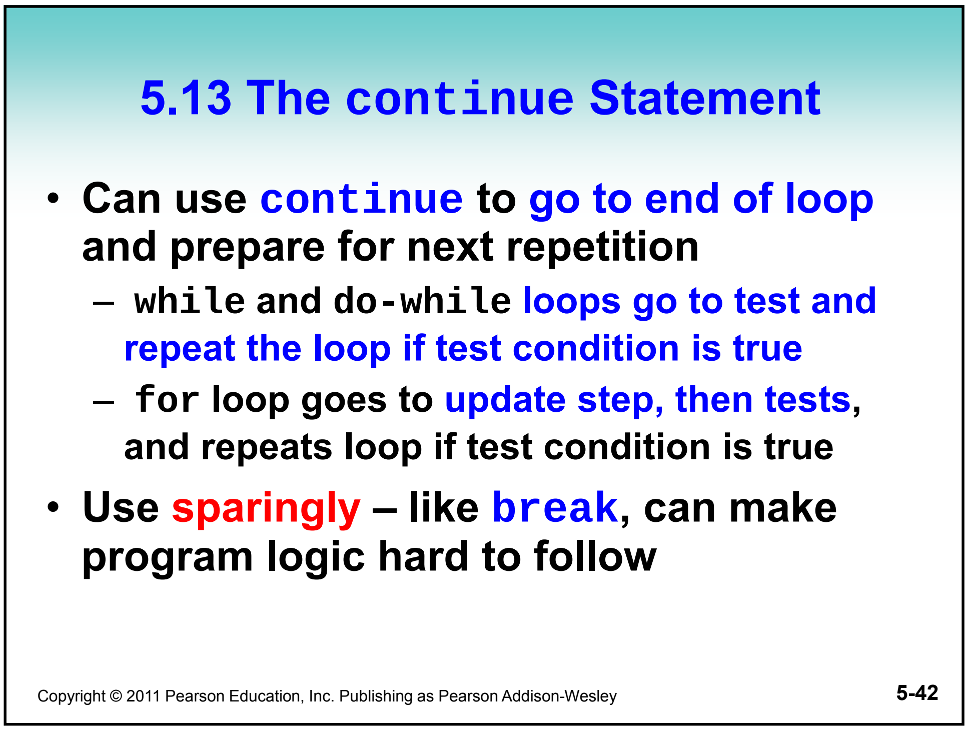
Notes on Nested Loops

- Inner loop goes through all its repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops. In previous example, inner loop repeats 9 times



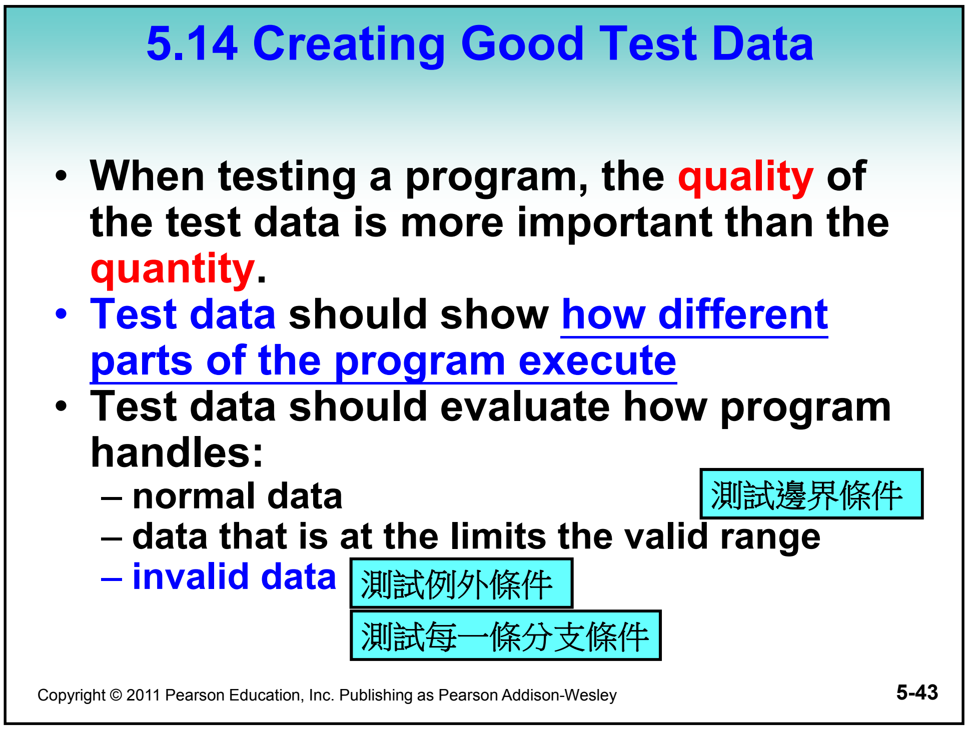
5.12 Breaking Out of a Loop

- Can use `break` to terminate execution of a loop
- Use sparingly – it could make your code harder to understand
- When used in an inner loop, terminates that loop only and returns to the outer loop 在多層迴圈中，一個 **Break** 指令只跳出一 層



5.13 The `continue` Statement

- Can use `continue` to go to end of loop and prepare for next repetition
  - `while` and `do-while` loops go to test and repeat the loop if test condition is true
  - `for` loop goes to update step, then tests, and repeats loop if test condition is true
- Use sparingly – like `break`, can make program logic hard to follow



5.14 Creating Good Test Data

- When testing a program, the quality of the test data is more important than the quantity.
- Test data should show <u>how different parts of the program execute</u>
- Test data should evaluate how program handles:
  - normal data
  - data that is at the limits the valid range
  - invalid data

測試邊界條件

測試例外條件

測試每一條分支條件

Chapter 6: Functions

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Topics

6.1 Modular Programming
6.2 Defining and Calling Functions
6.3 Function Prototypes
6.4 Sending Data into a Function
6.5 Passing Data by Value
6.6 The `return` Statement
6.7 Returning a Value from a Function
6.8 Returning a Boolean Value

Topics (continued)

6.9 Using Functions in a Menu-Driven Program
6.10 Local and Global Variables
6.11 Static Local Variables
6.12 Default Arguments
6.13 Using Reference Variables as Parameters
6.14 Overloading Functions
6.15 The `exit()` Function
6.16 Stubs and Drivers

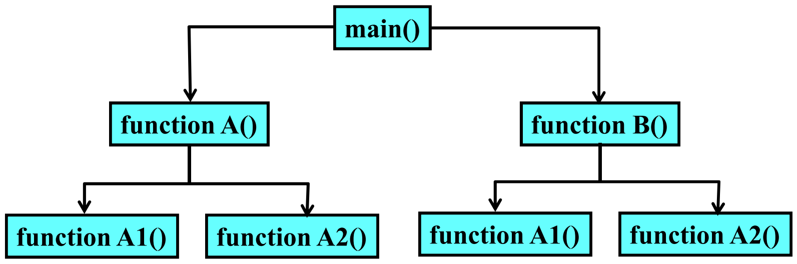
6.1 Modular Programming

- **Modular programming**: breaking a program up into smaller, manageable functions or modules
- **Function**: a collection of statements to perform a specific task
- **Motivation for modular programming**
  - Simplifies the process of writing programs
  - Improves maintainability of programs

Why Functions?

Modular Programming Style:
- **A program is divided into a number of modules (or functions), so that similar operations can be done by the same functions (with perhaps different data sets**

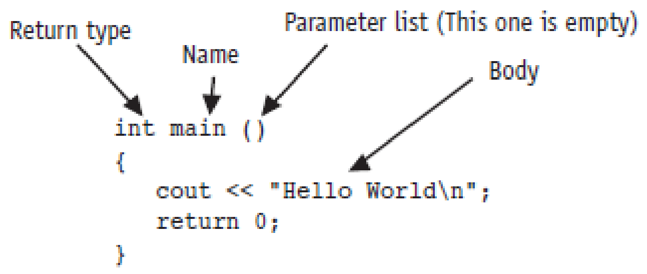
Main() is the caller (or parent) of A() as the callee (or child) function
A() is the caller (or parent) function of A1() as the callee (or child) function



6.2 Defining and Calling Functions

- **Function call: statement that causes a function to execute**
- **Function definition: statements that make up a function**

Function Definition



```
int main ()
{
    cout << "Hello World\n";
    return 0;
}
```

Function Definition (Details)

- **Definition includes**

  name: name of the function. Function names follow same rules as variable names

  parameter list: variables that hold the values passed to the function

  body: statements that perform the function's task

  return type: data type of the value the function returns to the part of the program that called it

Function Header

- The **function header** consists of
  - the function *return type*
  - the function *name*
  - the function *parameter list*
- Example:

```
int main()
```

- Note: no ; at the end of the header

Function Return Type

- If a function returns a value, the type of the value must be indicated

```
int main()
```

- If a function does not return a value, its return type is void

```
void printHeading()
{
   cout << "\tMonthly Sales\n";
}
```

Calling a Function

- To call a function, use the function name followed by ( ) and ;

  `printHeading();`

- When a function is called, the program executes the body of the function
- After the function terminates, execution resumes in the calling module at the point of call

Calling a Function

- `main` is automatically called when the program starts
- `main` can call any number of functions
- Functions can call other functions

6.3 Function Prototypes

The compiler must know the following about a function before it is called

- name
- return type
- number of parameters
- data type of each parameter

Function Prototypes

Ways to notify the compiler about a function before a call to the function:

- Place function definition before calling function
- Use a function prototype (similar to the heading of the function
  - Heading: `void printHeading()`
  - Prototype: `void printHeading();`

Prototype Notes

- Place prototypes near top of program
- Program must include either prototype or full function definition before any call to the function, otherwise a compiler error occurs
- When using prototypes, function definitions can be placed in any order in the source file. Traditionally, `main` is placed first.

6.4 Sending Data into a Function

- Can pass values into a function at the time of function call

```
c = sqrt(a*a + b*b);
```

- Values passed to function are arguments
- Variables in function that hold values passed as arguments are parameters
- Alternate names:
  - argument: actual argument, actual parameter 真實的值
  - parameter: formal argument, formal parameter 形式

Parameters, Prototypes, and Function Headings

- For each function argument,
  - the prototype must include the data type of each parameter in its ( )

```
void evenOrOdd(int);      //prototype
```

  - the heading must include a declaration, with variable type and name, for each parameter in its ( )

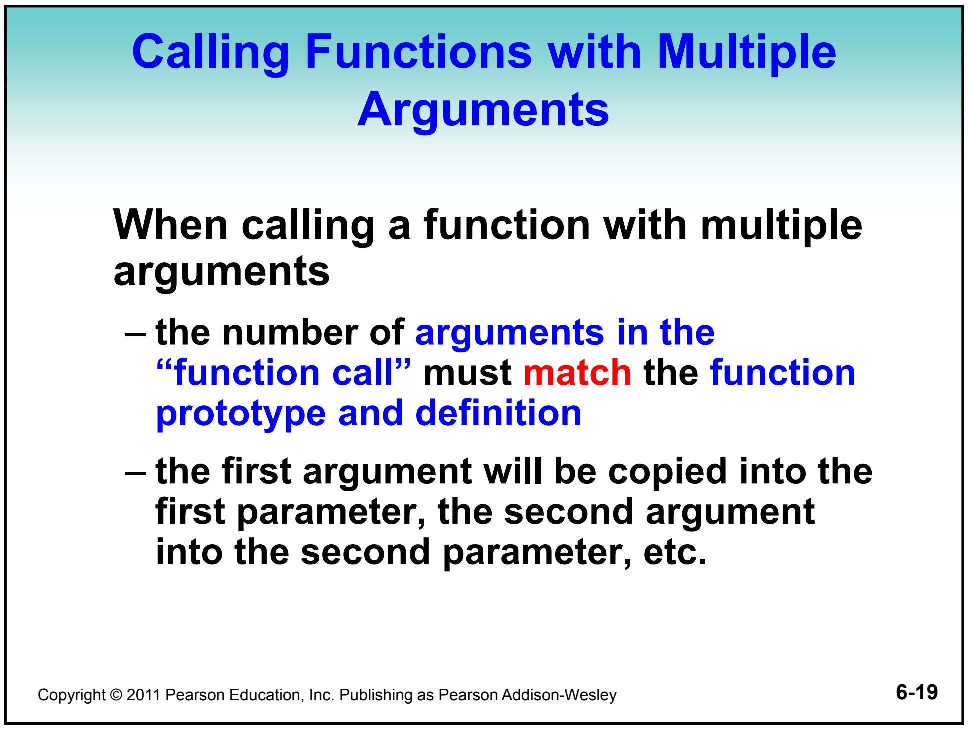
```
void evenOrOdd(int num)   //heading
```

- The parent function that calls the above function would look like this:

```
   evenOrOdd(val);  //call
```

Notes on Making A Function Call

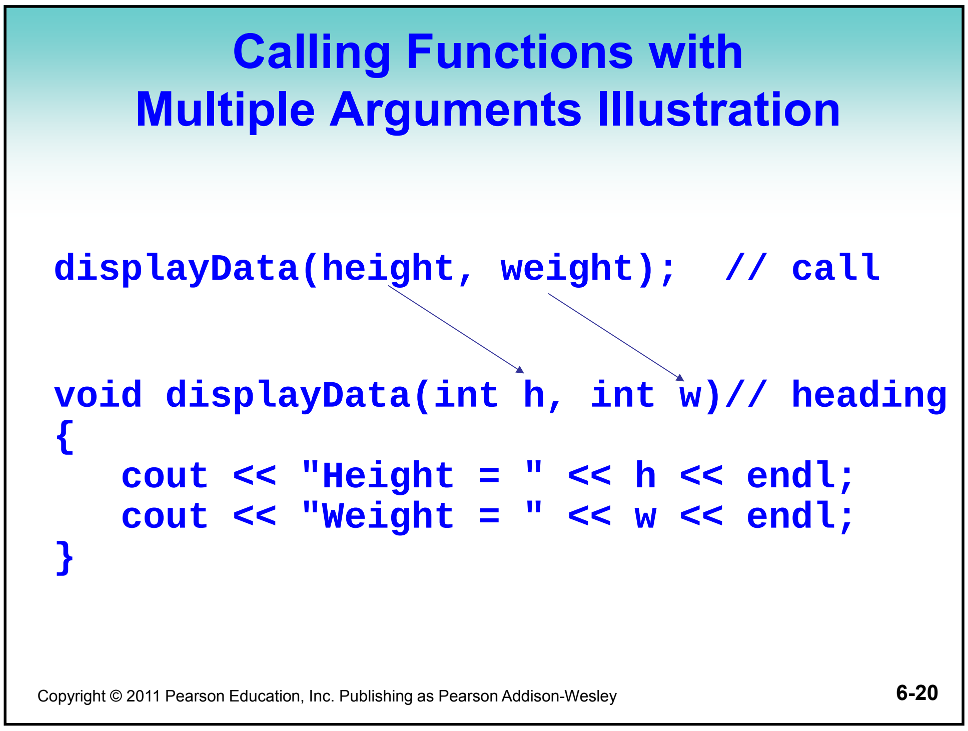
- Value of argument is <u>copied</u> into parameter when a function is called
- Function can have > 1 parameter
- There must be a data type listed in the prototype ( ) and an argument declaration in the function heading ( ) for each parameter
- Arguments will be promoted/demoted as necessary to match parameters



Calling Functions with Multiple Arguments

When calling a function with multiple arguments

- the number of arguments in the “function call” must match the function prototype and definition
- the first argument will be copied into the first parameter, the second argument into the second parameter, etc.



Calling Functions with Multiple Arguments Illustration

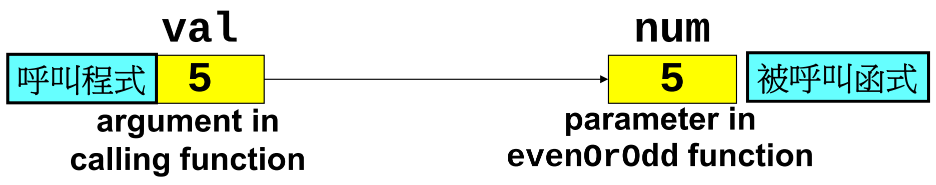
```
displayData(height, weight);  // call


void displayData(int h, int w)// heading
{
   cout << "Height = " << h << endl;
   cout << "Weight = " << w << endl;
}
```

6.5 Passing Data by Value

- **Pass by value**: when argument is passed to a function, a copy of its value is placed in the parameter
- A function cannot access the original argument (in its parent function)
- Changes to the parameter in the (child) function do not affect the value of the argument in the calling (or parent) function

Passing Data to Parameters by Value

- Example: `int val = 5;`
  `evenOrOdd(val);`



- `evenOrOdd` can change variable `num`, but it will have no effect on variable `val`

6.6 The `return` Statement

- Used to end execution of a function
- Can be placed anywhere in a function
  - Any statements that follow the `return` statement will not be executed
- Can be used to prevent abnormal termination of program
- Without a `return` statement, the function ends at its last }

6.7 Returning a Value From a Function

- `return` statement can be used to return a value from the function to the module that made the function call
- Prototype and definition must indicate data type of return value (not `void`)
- The parent function should use the "returned" value from a child function, *e.g.*,
  - assign it to a variable
  - send it to `cout`
  - use it in an arithmetic computation
  - use it in a relational expression

Returning a Value – the `return` Statement

- Format: `return expression;`
- *`expression`* may be a variable, a literal value, or an expression.
- *`expression`* should be of the same data type as the declared return type of the function (will be converted if not)

6.8 Returning a Boolean Value

- Function can return `true` or `false`
- Declare return type in function prototype and heading as `bool`
- Function body must contain `return` statement(s) that return `true` or `false`
- Calling function can use return value in a relational expression

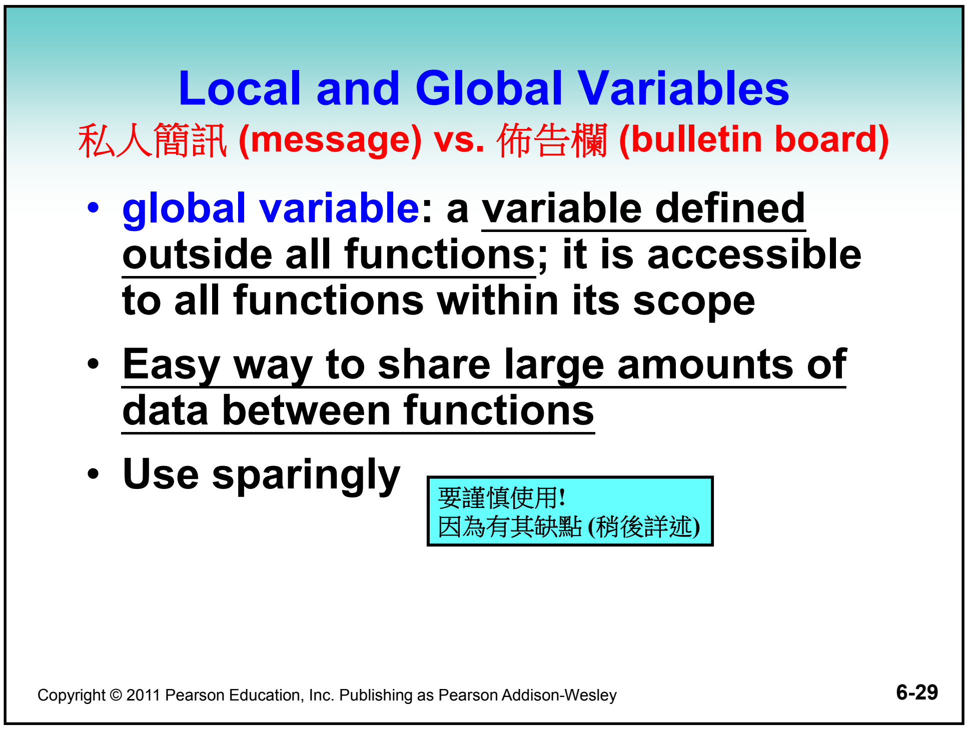
Boolean return Example

```
bool isValid(int);          // prototype
bool isValid(int val)       // heading
{
   int min = 0, max = 100;
   if (val >= min && val <= max)
      return true;
   else
      return false;
}
if (isValid(score))         // call
   …
```

6.10 Local and Global Variables

Check Appendix 1

- local variable: defined within a function or block; accessible only within the function or block
- Other functions and blocks can define other variables with the same name
- When a function is called, local variables in the calling function are not accessible from within the called function

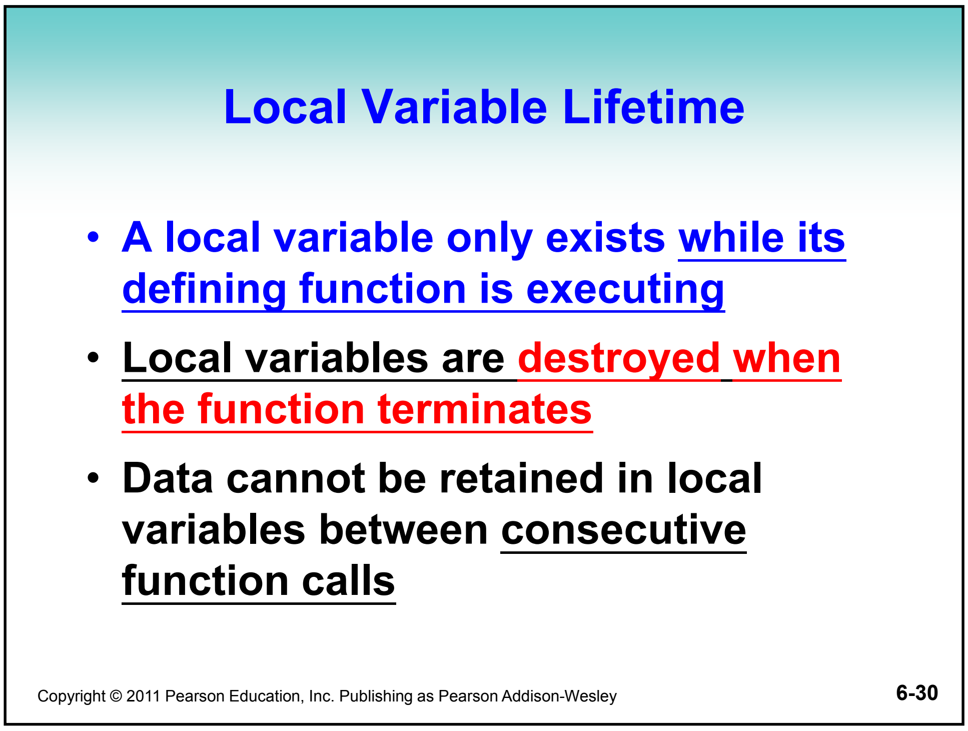


Local and Global Variables

私人簡訊 (message) vs. 佈告欄 (bulletin board)

- global variable: a variable defined outside all functions; it is accessible to all functions within its scope
- Easy way to share large amounts of data between functions
- Use sparingly

要謹慎使用!
因為有其缺點 (稍後詳述)



Local Variable Lifetime

- A local variable only exists while its defining function is executing
- Local variables are destroyed when the function terminates
- Data cannot be retained in local variables between consecutive function calls

Initializing Local and Global Variables

- Local variables must be initialized by the programmer
- Global variables are initialized to `0` (numeric) or `NULL` (character) when the variable is defined

Global Variables – Why Use Sparingly?

Global variables make:

- Programs that are hard to understand
- Programs that are difficult to debug
- Functions that cannot easily be re-used in other programs

Local and Global Variable Names

- Local variables can have the same names as global variables
- When a function contains a local variable that has the same name as a global variable, the global variable is unavailable from within the function. The local definition "hides" or "shadows" the global definition.

6.11 Static Local Variables

- Local variables
  - Only exist while the function is executing
  - Are redefined each time function is called
  - Lose their contents when function terminates
- `static` local variables
  - Are defined with key word `static`

    ```
    static int counter;
    ```
  - Are defined and initialized only the first time the function is executed
  - Retain their contents between consecutive function calls

6.12 Default Arguments

- Values passed automatically if arguments are missing from the function call
- Must be a constant declared in prototype

```
void evenOrOdd(int = 0);
```

- Multi-parameter functions may have default arguments for some or all of them

```
int getSum(int, int=0, int=0);
```

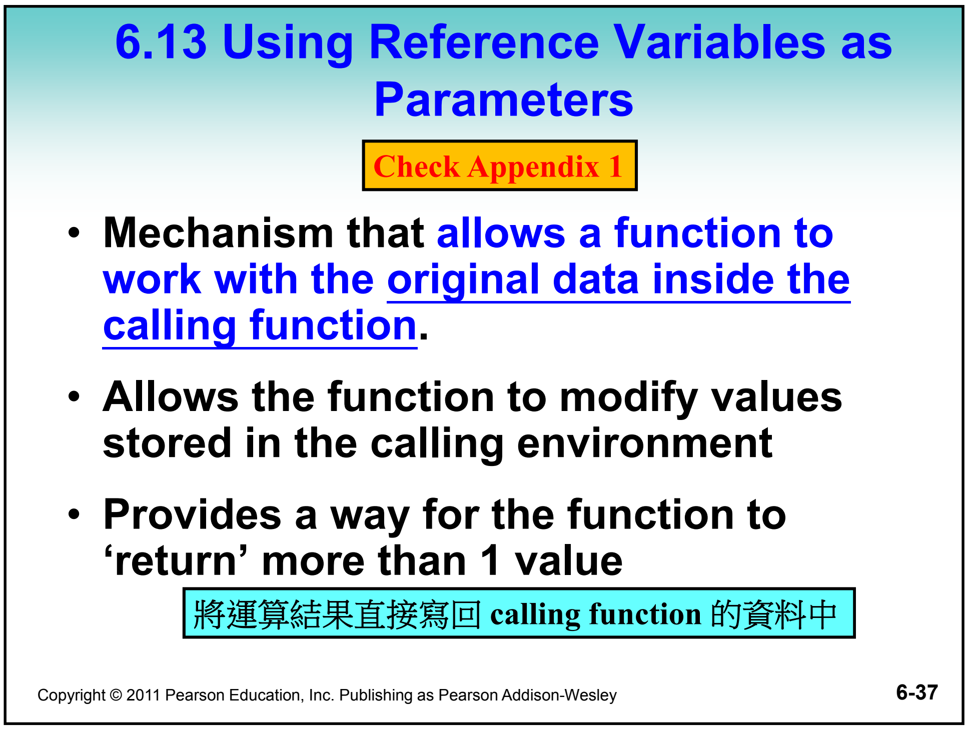
Default Arguments

- If not all parameters to a function have default values, the ones without defaults must be declared first in the parameter list

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // wrong!
```

- When an argument is omitted from a function call, all arguments after it must also be omitted

```
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);   // wrong!
```

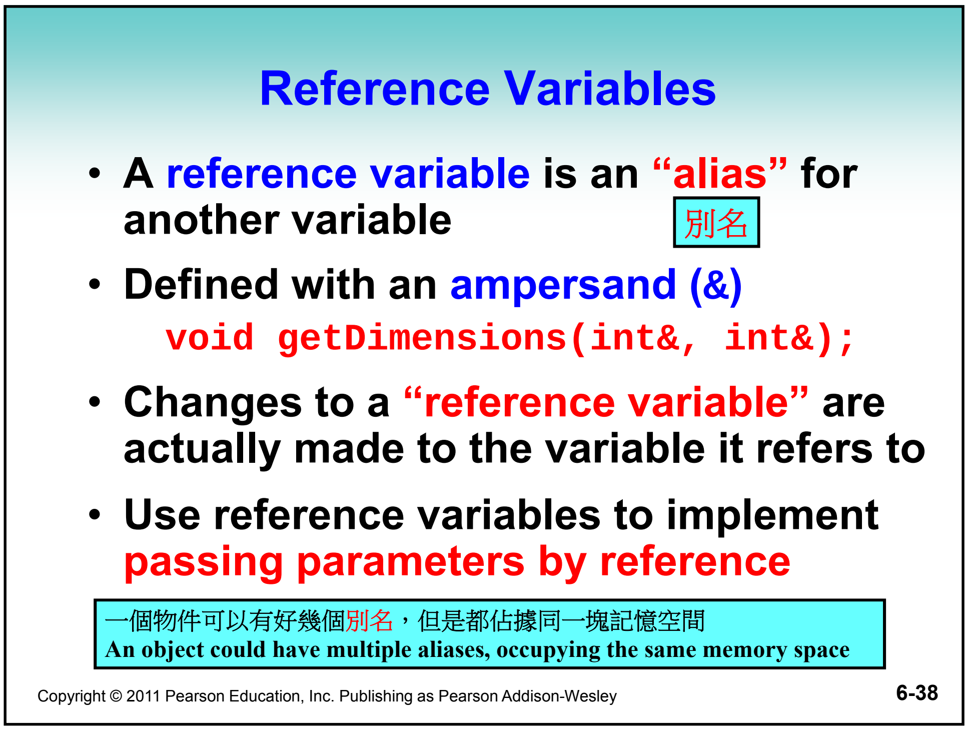


6.13 Using Reference Variables as Parameters

Check Appendix 1

- Mechanism that allows a function to work with the original data inside the calling function.
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to ‘return’ more than 1 value

將運算結果直接寫回 calling function 的資料中



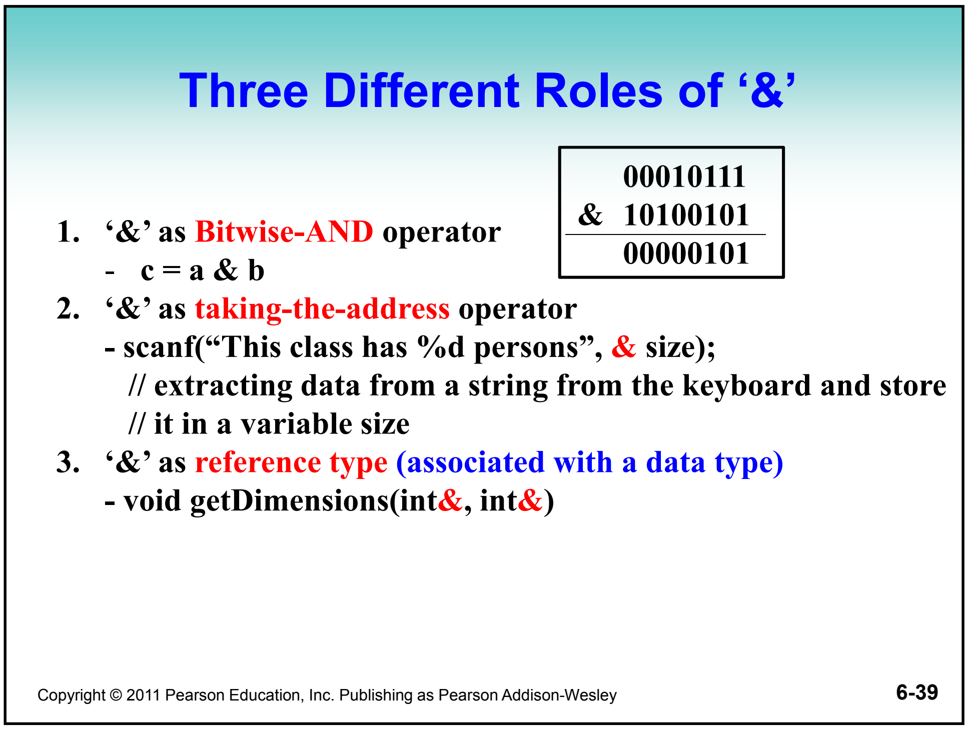
Reference Variables

- A reference variable is an “alias” for another variable 別名
- Defined with an ampersand (&)

```
void getDimensions(int&, int&);
```

- Changes to a “reference variable” are actually made to the variable it refers to
- Use reference variables to implement passing parameters by reference

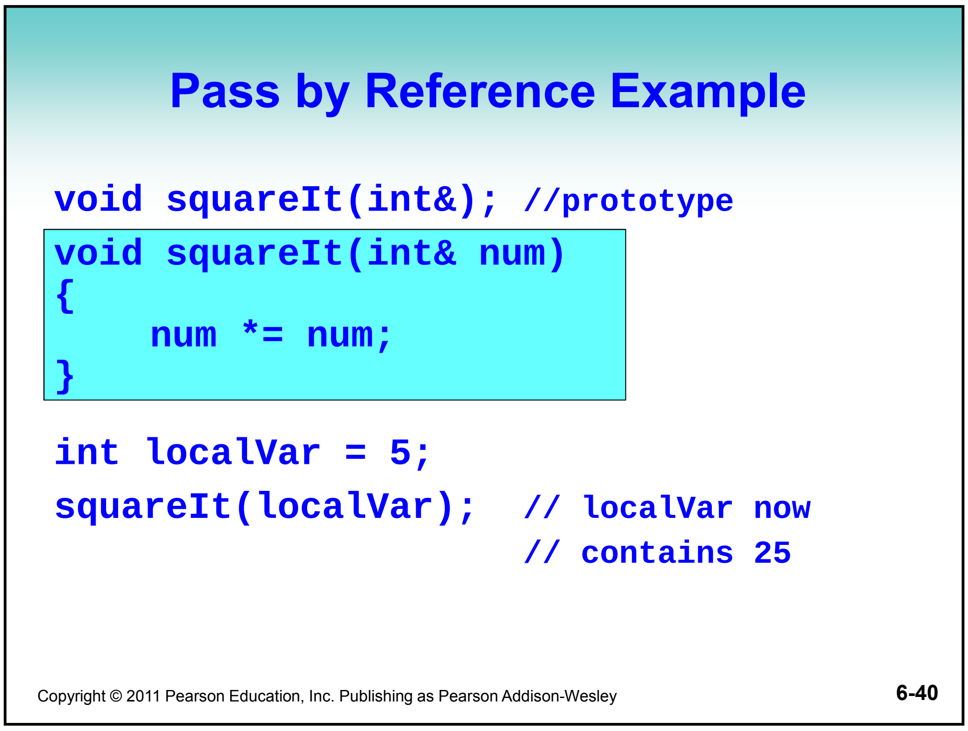
一個物件可以有好幾個別名，但是都佔據同一塊記憶空間
An object could have multiple aliases, occupying the same memory space



Three Different Roles of '&'

```
   00010111
&  10100101
-----------
   00000101
```

1. '&' as Bitwise-AND operator
   - c = a & b
2. '&' as taking-the-address operator
   - scanf("This class has %d persons", & size);
     // extracting data from a string from the keyboard and store
     // it in a variable size
3. '&' as reference type (associated with a data type)
   - void getDimensions(int&, int&)



Pass by Reference Example

```
void squareIt(int&); //prototype
void squareIt(int& num)
{
    num *= num;
}

int localVar = 5;
squareIt(localVar);   // localVar now
                      // contains 25
```

Reference Variable Notes

- Each reference parameter must contain &
- Argument passed to reference parameter must be a variable (cannot be an expression or constant)
- Use only when appropriate, such as when the function must input or change the value of the argument passed to it
- Files (*i.e.*, file stream objects) should be passed by reference

6.14 Overloading Functions

- Overloaded functions are two or more functions that have the same name, but different parameter lists
- Can be used to create functions that perform the same task, but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter list

Signature

Overloaded Functions Example

If a program has these overloaded functions,

```
void getDimensions(int);           // 1
void getDimensions(int, int);      // 2
void getDimensions(int, float);    // 3
void getDimensions(double, double);// 4
```

then the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);             // 1
getDimensions(length, width);      // 2
getDimensions(length, height);     // 3
getDimensions(height, base);       // 4
```

6.15 The `exit()` Function

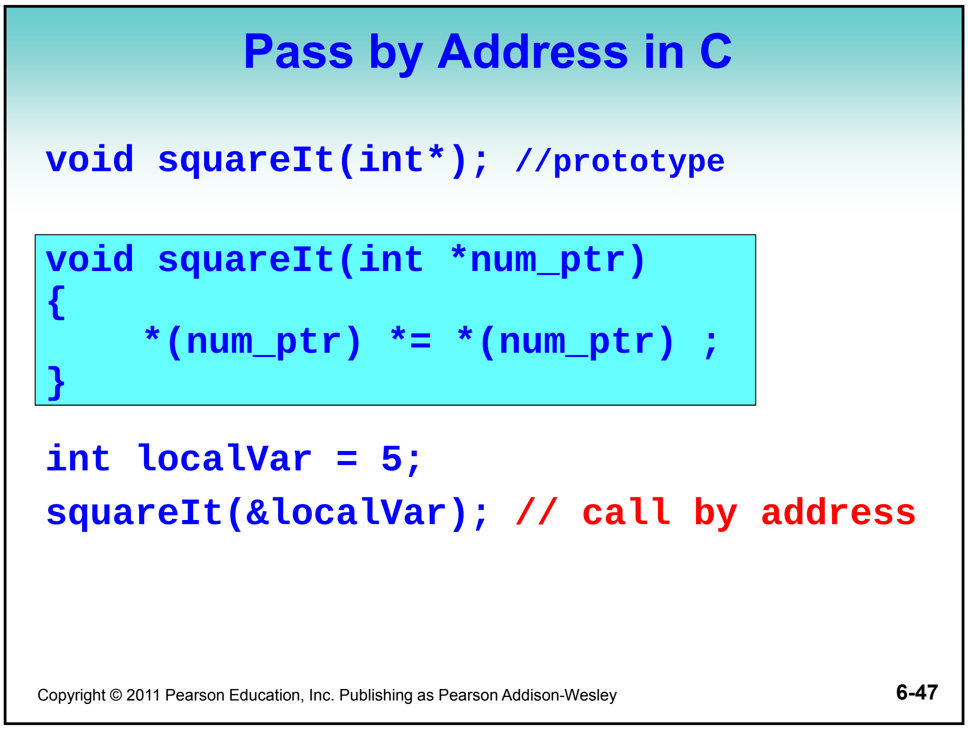
- Terminates execution of the entire program
- Can be called from any function
- Can pass a value to operating system to indicate status of program execution
- Usually used for abnormal termination of program
- Requires `cstdlib` header file
- Use carefully

`exit()` – Passing Values to Operating System

- Use an integer value to indicate program status
- Often, 0 means successful completion, non-zero indicates a failure condition
- Can use named constants defined in `cstdlib`:
  - `EXIT_SUCCESS` and
  - `EXIT_FAILURE`

6.16 Stubs and Drivers

- Stub: dummy function in place of actual function
- Usually displays a message indicating it was called. May also display parameters
- Driver: function that tests a function by calling it
- Stubs and drivers are useful for testing and debugging program logic and design



Pass by Address in C

```
void squareIt(int*); //prototype

void squareIt(int *num_ptr)
{
    *(num_ptr) *= *(num_ptr) ;
}

int localVar = 5;
squareIt(&localVar); // call by address
```

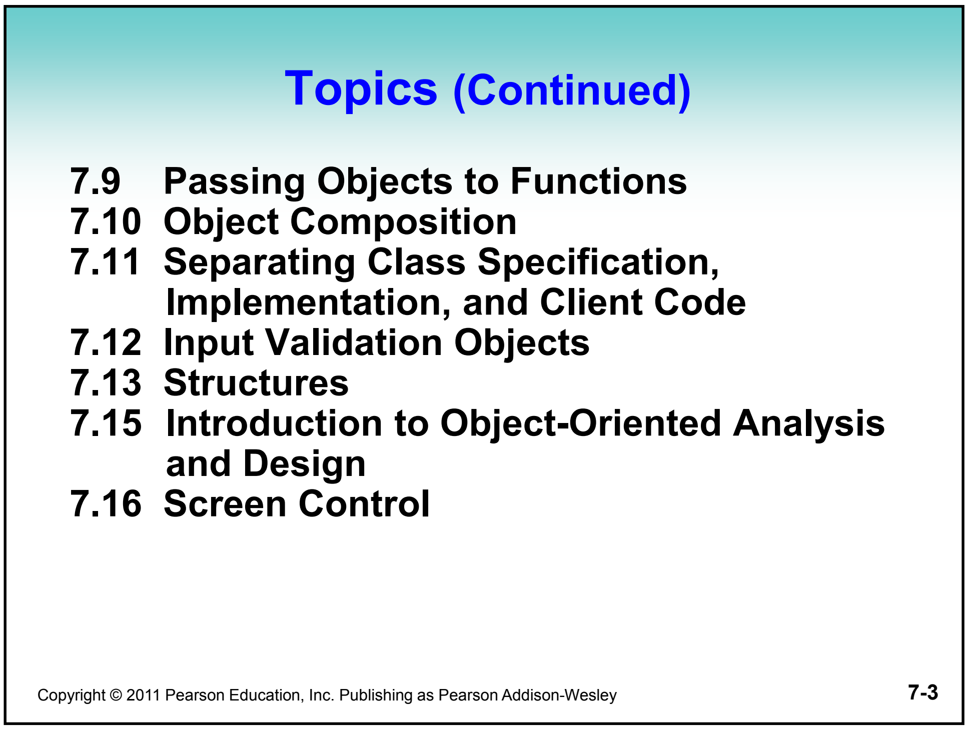
Chapter 7: Introduction to Classes and Objects

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

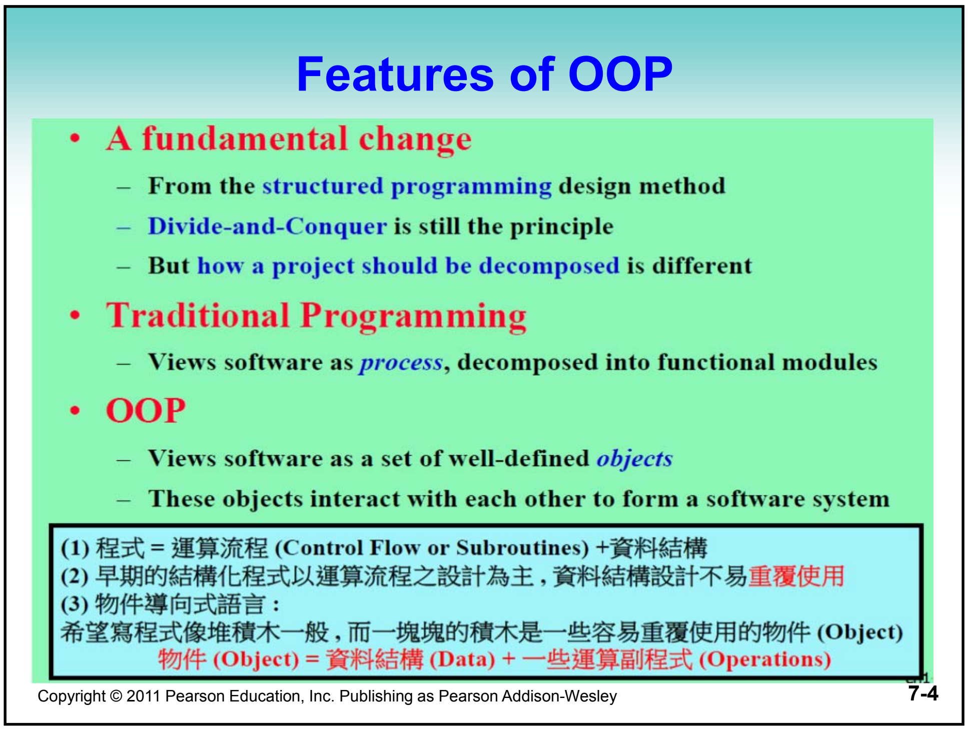
Topics

7.1 Abstract Data Types
7.2 Object-Oriented Programming
7.3 Introduction to Classes
7.4 Introduction to Objects
7.5 Defining Member Functions
7.6 Constructors
7.7 Destructors
7.8 Private Member Functions



Topics (Continued)

7.9 Passing Objects to Functions
7.10 Object Composition
7.11 Separating Class Specification, Implementation, and Client Code
7.12 Input Validation Objects
7.13 Structures
7.15 Introduction to Object-Oriented Analysis and Design
7.16 Screen Control



Features of OOP

- A fundamental change
  - From the **structured programming** design method
  - **Divide-and-Conquer** is still the principle
  - But **how a project should be decomposed** is different
- Traditional Programming
  - Views software as *process*, decomposed into functional modules
- OOP
  - Views software as a set of well-defined *objects*
  - These objects interact with each other to form a software system

(1) 程式 = 運算流程 (Control Flow or Subroutines) +資料結構
(2) 早期的結構化程式以運算流程之設計為主，資料結構設計不易重覆使用
(3) 物件導向式語言：
希望寫程式像堆積木一般，而一塊塊的積木是一些容易重覆使用的物件 (Object)
物件 (Object) = 資料結構 (Data) + 一些運算副程式 (Operations)

7.1 Abstract Data Types

- Programmer-created data types that specify
  - legal values that can be stored
  - operations that can be done on the values
- The user of an abstract data type (ADT) does not need to know any implementation details (*e.g.*, how the data is stored or how the operations on it are carried out)

Abstraction and Data Types

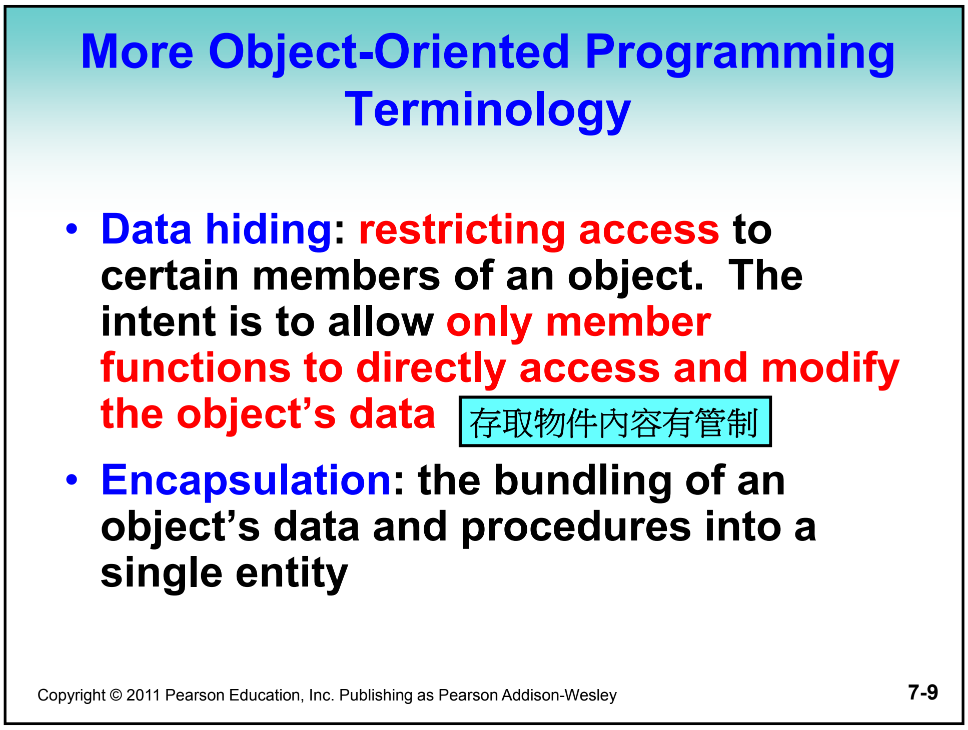
- Abstraction: a definition that captures general characteristics without details
  - An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral
- Data Type: defines the kind of values that can be stored and the operations that can be performed on it

7.2 Object-Oriented Programming

- Procedural programming uses variables to store data, focuses on the processes/ functions that occur in a program. Data and functions are separate and distinct.
- Object-oriented programming is based on objects that encapsulate the data and the functions that operate on it.

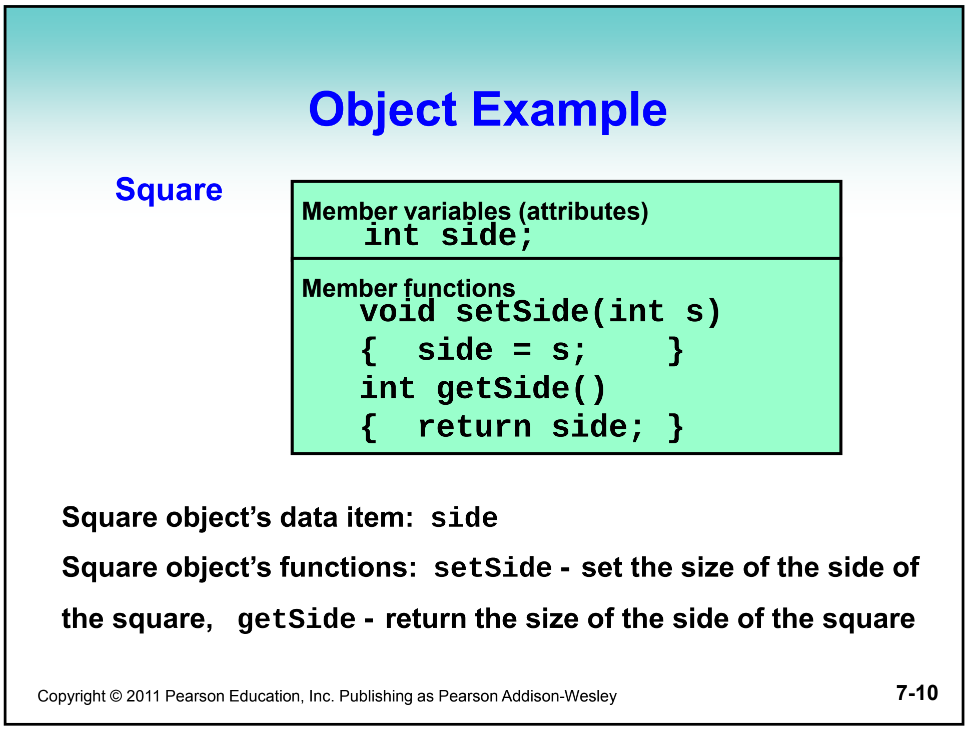
Object-Oriented Programming Terminology

- Object: software entity that combines data and functions that act on the data in a single unit
- Attributes: the data items of an object, stored in member variables
- Member functions (methods): procedures/ functions that act on the attributes of the class



More Object-Oriented Programming Terminology

- **Data hiding**: restricting access to certain members of an object. The intent is to allow only member functions to directly access and modify the object's data 存取物件內容有管制
- **Encapsulation**: the bundling of an object's data and procedures into a single entity



Object Example

Square

Member variables (attributes)

```
    int side;
```

Member functions

```
    void setSide(int s)
    {  side = s;    }
    int getSide()
    {  return side; }
```

Square object's data item: `side`

Square object's functions: `setSide` - set the size of the side of the square, `getSide` - return the size of the side of the square

7.3 Introduction to Classes

- **Class**: a **programmer-defined data type used** to define objects
- **It is a pattern for creating objects**
- **Class declaration format**:

```
class className
{
    declaration;
    declaration;
};
```

Notice the required ;

Access Specifiers
- *public* or *private*

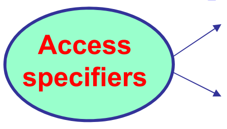
- **Used to control access to members of the class.**
- **Each member is declared to be either**

  `public`: **can be accessed by functions outside of the class**

  or

  `private`: **can only be called by or accessed by functions that are members of the class**

Class Example

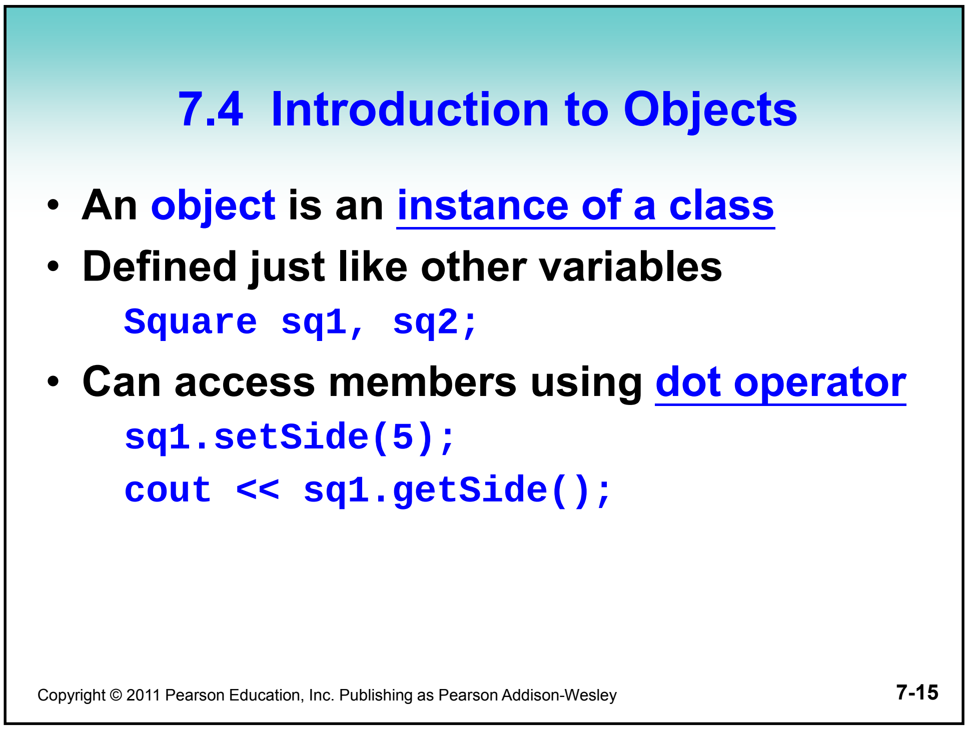


```
class Square
{
  private:
    int side;
  public:
    void setSide(int s)
    { side = s; }
    int getSide()
    { return side; }
};
```

More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is `private`

以實踐【資料包裹性】
or Data Encapsulation



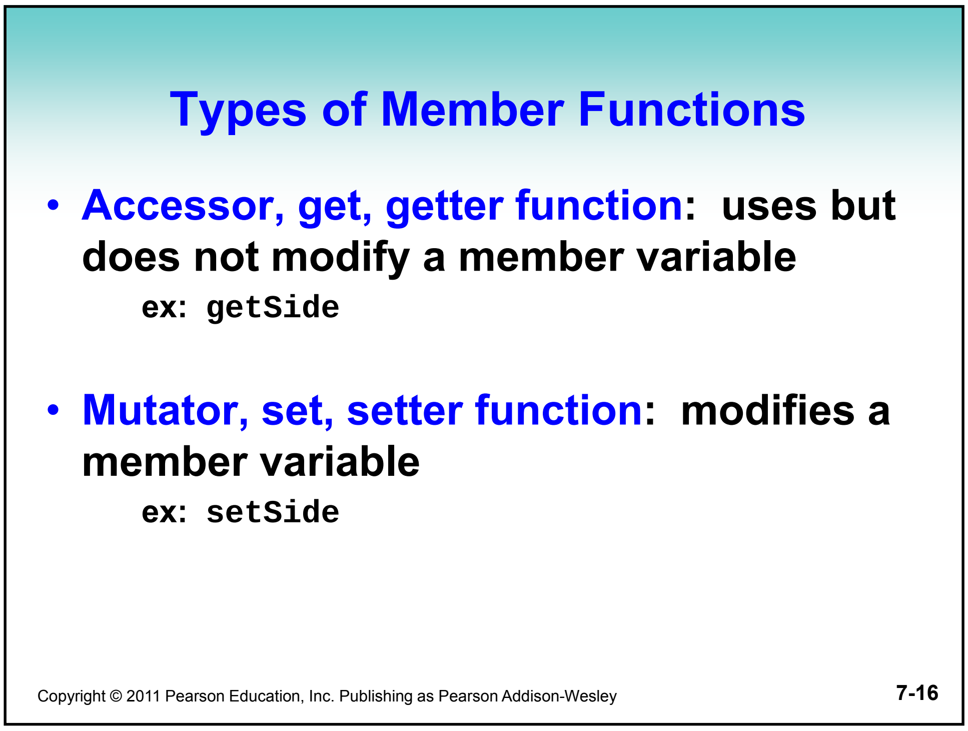
7.4 Introduction to Objects

- An **object** is an **instance of a class**
- Defined just like other variables

```
Square sq1, sq2;
```

- Can access members using **dot operator**

```
sq1.setSide(5);
cout << sq1.getSide();
```



Types of Member Functions

- **Accessor, get, getter function**: uses but does not modify a member variable

  ex: `getSide`

- **Mutator, set, setter function**: modifies a member variable

  ex: `setSide`

7.5 Defining Member Functions

- Member functions are part of a class declaration
- Can place entire function definition inside the class declaration 函式原型、定義二合一

  or
- Can place just the prototype inside the class declaration and write the function definition after the class 函式原型、定義分開

函式原型 (框架而已): **Function Prototype**
函式定義 (內容): **Function Body**

Defining Member Functions Inside the Class Declaration

- Member functions defined inside the class declaration are called inline functions
- Only very short functions, like the one below, should be inline functions

```
int getSide()
{ return side; }
```

Inline Member Function Example

```
class Square
{
   private:
      int side;
   public:
      void setSide(int s)
      { side = s; }
      int getSide()
      { return side; }
};
```

inline functions

Defining Member Functions After the Class Declaration

- Put a function prototype in the class declaration
- In the function definition, precede function name with class name and scope resolution operator (::)

```
int Square::getSide()
{
  return side;
}
```

當一個 **member function** 的內容定義在 **class** 宣告外的話，就必須在 **member function** 的名字前加 **Scope Resolution Operator**

Conventions and a Suggestion

Conventions:

- Member variables are usually `private`
- Accessor and mutator functions are usually `public`
- Use ‘get’ in the name of accessor functions, ‘set’ in the name of mutator functions

Suggestion: calculate values to be returned in accessor functions when possible, to minimize the potential for stale data

Tradeoffs of Inline vs. Regular Member Functions

- When a regular function is called, control passes to the called function
  - the compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into the program in place of the call when the program is compiled
  - larger executable program, but
  - less function call overhead, possibly faster execution

Inline functions ➔ Trade code size for speed!

7.6 Constructors

Constructor 是一個 member function，其名字與 class name 一樣!

- A **constructor** is a member function that is <u>used to initialize data members of a class</u>
- Is called automatically when an object of the class is created
- Must be a `public` member function
- Must be named the same as the class
- Must have no return type

Constructor – 2 Examples

Inline:

```
class Square
{
   . . .
   public:
      Square(int s)
      { side = s; }
   . . .
};
```

Declaration outside the class:

```
 Square(int);
//prototype in class

Square::Square(int s)
{
   side = s;
}
```

Overloading Constructors

- A class can have more than 1 constructor
- Overloaded constructors in a class must have different parameter lists

```
Square();
Square(int);
```

The Default Constructor

- Constructors can have any number of parameters, including none
- A default constructor is one that takes no arguments either due to
  - No parameters or
  - All parameters have default values
- If a class has any programmer-defined constructors, it must have a programmer-defined default constructor

Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square()          // default
    { side = 1; }     // constructor

    // Other member
    // functions go here
};
```

Has no parameters

Another Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square(int s = 1) // default
    { side = s; }     // constructor

    // Other member
    // functions go here
};
```

Has parameter but it has a default value

Invoking a Constructor

- To create an object using the default constructor, use no argument list and no ( )

```
Square square1;
```

- To create an object using a constructor that has parameters, include an argument list

```
Square square1(8);
```

7.7 Destructors

- Public member function automatically called when an object is destroyed
- Destructor name is ~*className*, *e.g.*, `~Square`
- Has no return type
- Takes no arguments
- Only 1 destructor is allowed per class (*i.e.*, it cannot be overloaded)

7. 8 Private Member Functions

- A `private` member function can only be called by another member function of the same class
- It is used for internal processing by the class, not for the use outside of the class

7.9 Passing Objects to Functions

- A class object can be passed as an argument to a function
- When passed by value, function makes a local copy of object. Original object in calling environment is unaffected by actions in function
- When passed by reference, function can use 'set' functions to modify the object.

Notes on Passing Objects

- Using a value parameter for an object can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the structure Better, but risky!
- To save space and time, while protecting data that should not be changed, use a `const` reference parameter Fast and Efficient and Safe for read-only arguments!

```
void showData(const Square &s)
                          // header
```

Returning an Object from a Function

- A function can return an object

```
Square initSquare();   // prototype
s1 = initSquare();     // call
```

- Function must define an object
  - for internal use
  - to use with `return` statement

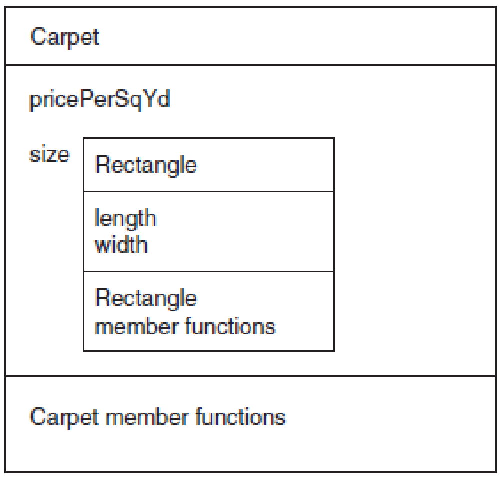
Returning an Object Example

```
Square initSquare()
{
  Square s;    // local variable
  int inputSize;
  cout << "Enter the length of side: ";
  cin >> inputSize;
  s.setSide(inputSize);
  return s;
}
```

7.10 Object Composition

- **Object composition occurs when an object is a member variable of another object.**
- **Often used to design complex objects whose members are simpler objects**
- **ex. (from book): Define a rectangle class. Then, define a carpet class and use a rectangle object as a member of a carpet object.**

Object Composition, cont.



7.11 Separating Class Specification, Implementation, and Client Code

Separating class declaration, member function definitions, and the program that uses the class into separate files is considered good design

Abstraction:
分離【規格】(header files) 和【實現方法】(cpp files)
Separate specification and implementation

Using Separate Files

- Place class declaration in a header file that serves as the class specification file. Name the file `classname.h` (for example, `Square.h`)
- Place member function definitions in a class implementation file. Name the file `classname.cpp` (for example, `Square.cpp`) This file should `#include` the class specification file.
- A client program (client code) that uses the class must `#include` the class specification file and be compiled and linked with the class implementation file.

#include <header files>, linked with <object files>

"Include" Guards

- to avoid multiple inclusions of the same header files

- Format:

```
#ifndef symbol_name
#define symbol_name
. . .  (normal contents of header file)
#endif
```

- *symbol_name* is usually the name of the header file, in all capital letters:

```
#ifndef SQUARE_H
#define SQUARE_H
. . .
#endif
```

7.12 Input Validation Objects

Classes can be designed to validate user input

– to ensure acceptable menu choice
– to ensure a value is in a range of valid values
– etc.

Class 內部可以做許多的 Sanity Check (或是 Validity Check)
➔ 及早偵測到錯誤的結果，避免 Nasty Bugs!

7.13 Structures

- Structure: C++ construct that allows multiple variables to be grouped together
- Structure Declaration Format:

```
struct structure name
{
  type1 field1;
  type2 field2;
     …
 typen fieldn;
};
```

Example struct Declaration

```
struct Student
{
   int studentID;
   string name;
   short year;
   double gpa;
};
```

structure tag

structure members

Notice the required ;

struct Declaration Notes

Keyword "*struct*" 與 "*class*" 一樣，but no "data encapsulation"
➔ That is, all members are all public!

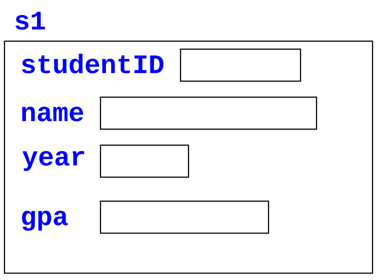
- `struct` names commonly begin with an uppercase letter
- The **structure name** is also called the **tag**
- Multiple fields of same type can be in a comma-separated list

```
string name,
       address;
```

Defining Structure Variables

- `struct` declaration does not allocate memory or create variables
- To define variables, use <u>structure tag</u> as <u>type name</u>

```
Student s1;
```



Accessing Structure Members

- Use the dot (.) operator to refer to members of `struct` variables

```
getline(cin, s1.name);
cin >> s1.studentID;
s1.gpa = 3.75;
```

- Member variables can be used in any manner appropriate for their data type

Displaying `struct` Members

To display the contents of a `struct` variable, you must display each field separately, using the dot operator

Wrong: 因為我們沒有 overload "operator<<" for *struct* "student"

```
cout << s1; // won't work!
```

Correct:

```
cout << s1.studentID << endl;
cout << s1.name << endl;
cout << s1.year << endl;
cout << s1.gpa;
```

Comparing `struct` Members

- Similar to displaying a `struct`, you cannot compare two `struct` variables directly:

因為我們沒有 overload "operator>= " for *struct* "student"

```
if (s1 >= s2) // won't work!
```

- Instead, compare member variables:

```
if (s1.gpa >= s2.gpa) // better
```

Initializing a Structure

Cannot initialize members in the structure declaration, because no memory has been allocated yet

struct 或 class 只是定義 data structure ，本身不是真實存在的物件，因此不可以有起始值

```
struct Student      // Illegal
{                   // initialization
   int studentID = 1145;
   string name = "Alex";
   short year = 1;
   float gpa = 2.95;
};
```

Initializing a Structure (continued)

- Structure members are initialized at the time a structure variable is created
- Can initialize a structure variable's members with either
  - an initialization list
  - a constructor

Using an Initialization List

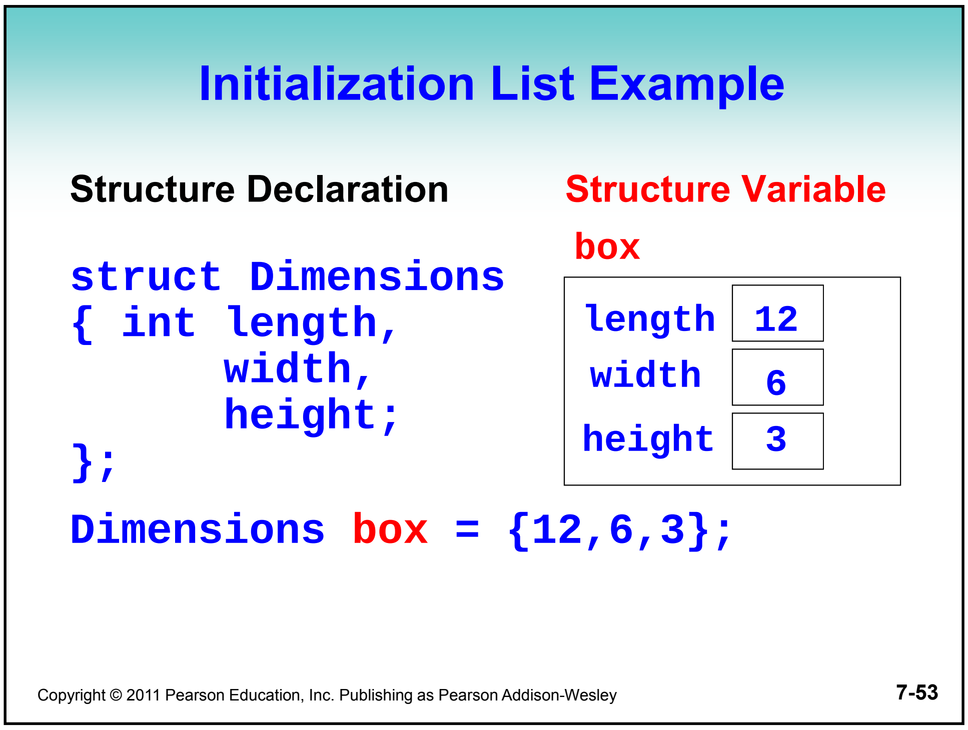
An initialization list is an ordered set of values, separated by commas and contained in { }, that provides initial values for a set of data members

```
{12, 6, 3}  // initialization list
            // with 3 values
```

More on Initialization Lists

- Order of list elements matters: First value initializes first data member, second value initializes second data member, etc.
- Elements of an initialization list can be constants, variables, or expressions

```
{12, W, L/W + 1} // initialization list
                 // with 3 items
```



Initialization List Example

Structure Declaration

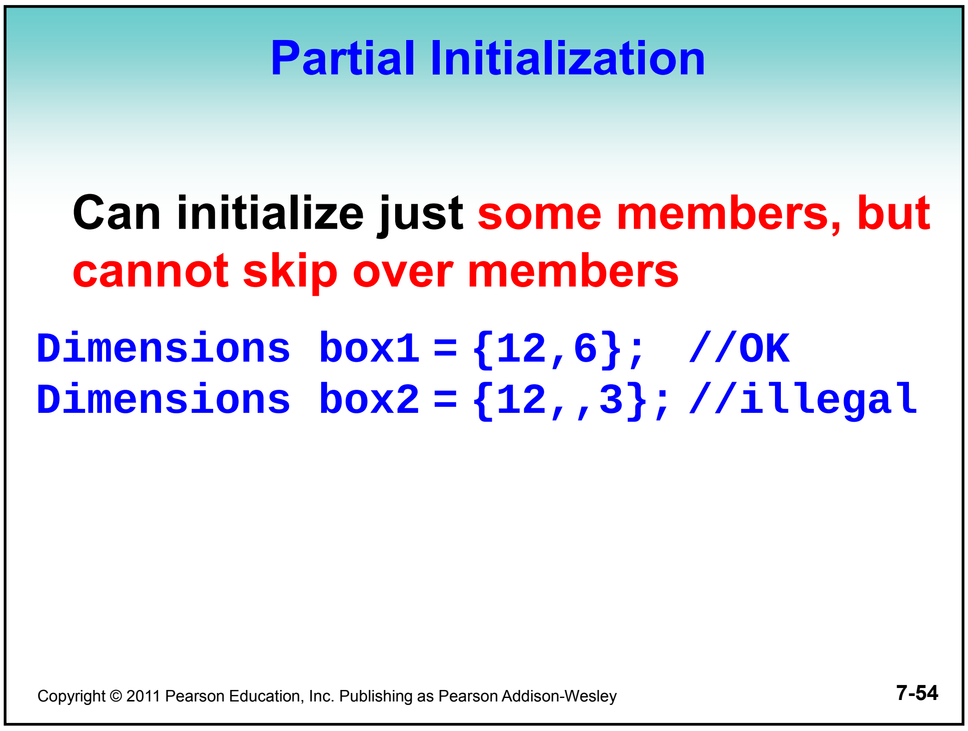
```
struct Dimensions
{ int length,
      width,
      height;
};
```

Structure Variable

box

length	12
width	6
height	3

```
Dimensions box = {12,6,3};
```



Partial Initialization

Can initialize just some members, but cannot skip over members

```
Dimensions box1 = {12,6};  //OK
Dimensions box2 = {12,,3}; //illegal
```

Problems with Initialization List

- Can't omit a value for a member without omitting values for all following members
  一旦跳掉了一個會員資料的起始，其後的會員資料也不能起始了…
- Does not work on most modern compilers if the structure contains any string objects
  C++ string 不work!
  - Will, however, work with C-string members

Using a Constructor to Initialize Structure Members

- Similar to a constructor for a class:
  - name is the same as the name of the struct
  - no return type
  - used to initialize data members
- It is normally written inside the `struct` declaration

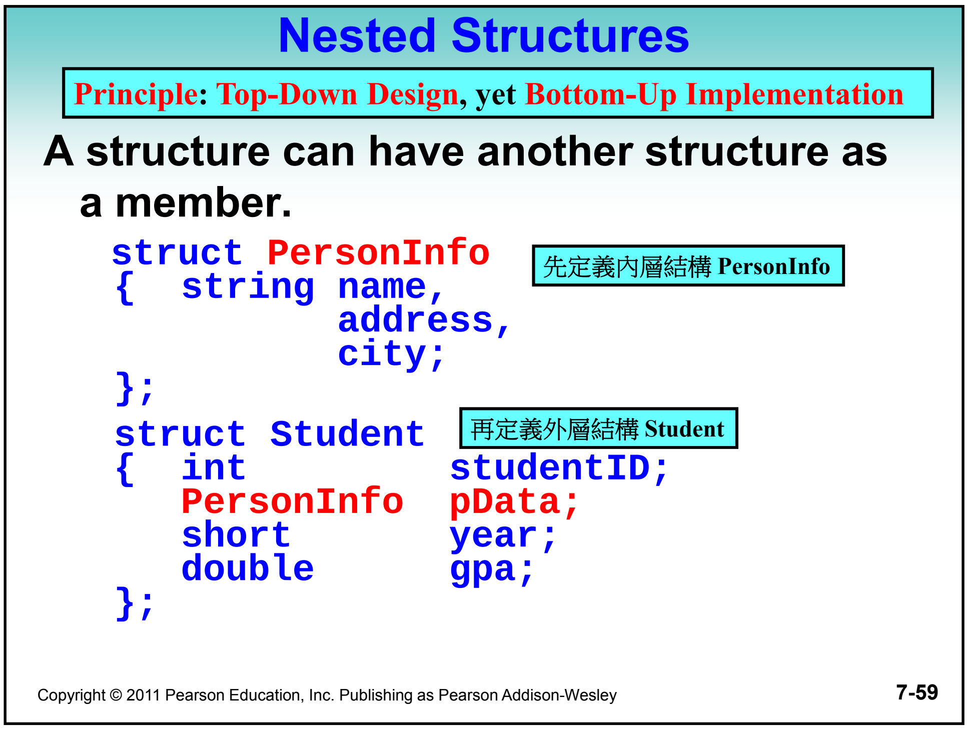
A Structure with a Constructor

```
struct Dimensions
{
  int length,
      width,
      height;
  // Constructor
  Dimensions(int L, int W, int H)
  {length = L; width = W; height = H;}
};
```

Passing Arguments to a Constructor

- Create a structure variable and follow its name with an argument list
- Example:

```
Dimensions box3(12, 6, 3);
```



Nested Structures

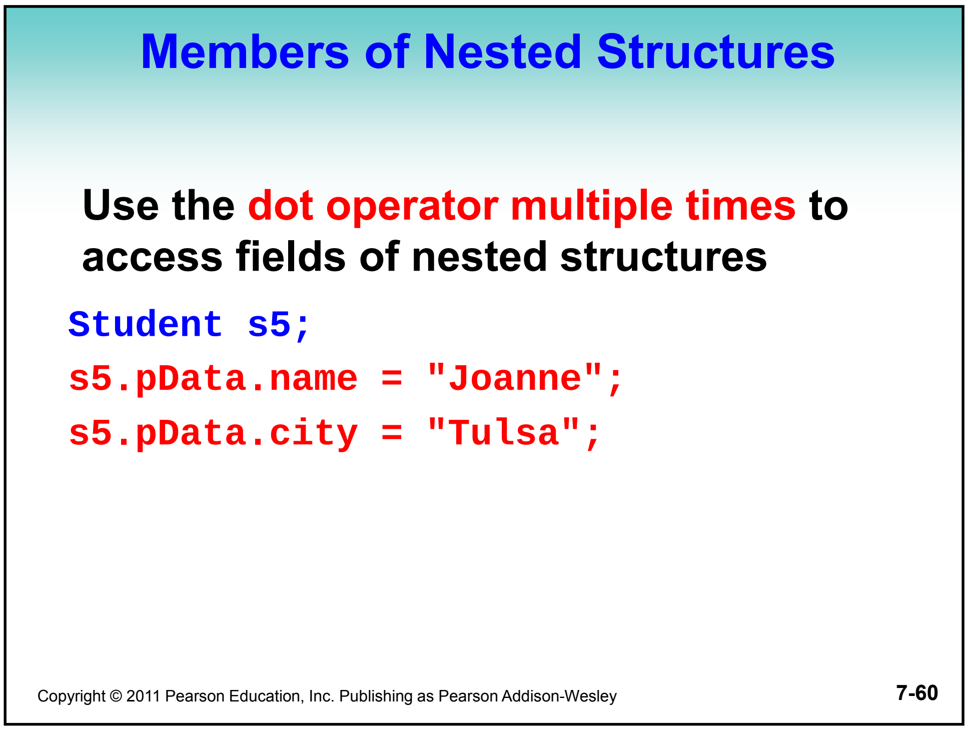
Principle: Top-Down Design, yet Bottom-Up Implementation

A structure can have another structure as a member.

```
struct PersonInfo
{  string name,
          address,
          city;
};
struct Student
{  int          studentID;
   PersonInfo   pData;
   short        year;
   double       gpa;
};
```

先定義內層結構 PersonInfo

再定義外層結構 Student



Members of Nested Structures

Use the dot operator multiple times to access fields of nested structures

```
Student s5;
s5.pData.name = "Joanne";
s5.pData.city = "Tulsa";
```

Structures as Function Arguments

- May pass members of `struct` variables to functions

  ```
  computeGPA(s1.gpa);
  ```

- May pass entire `struct` variables to functions

  ```
  showData(s5);
  ```

- Can use reference parameter if function needs to modify contents of structure variable

Notes on Passing Structures

- Using a value parameter for structure can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the structure
- To save space and time, while protecting structure data that should not be changed, use a `const` reference parameter

  ```
  void showData(const Student &s)
                            // header
  ```

Returning a Structure from a Function

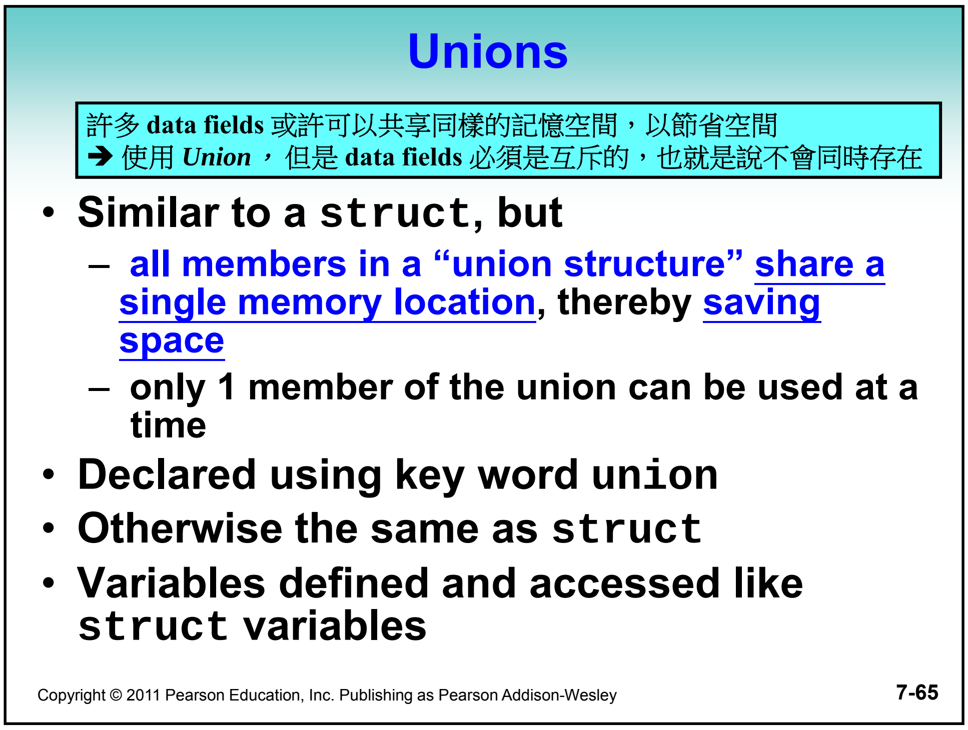
- Function can return a `struct`

```
Student getStuData();   // prototype
s1 = getStuData();      // call
```

- Function must define a local structure variable
  - for internal use
  - to use with `return` statement

Returning a Structure Example

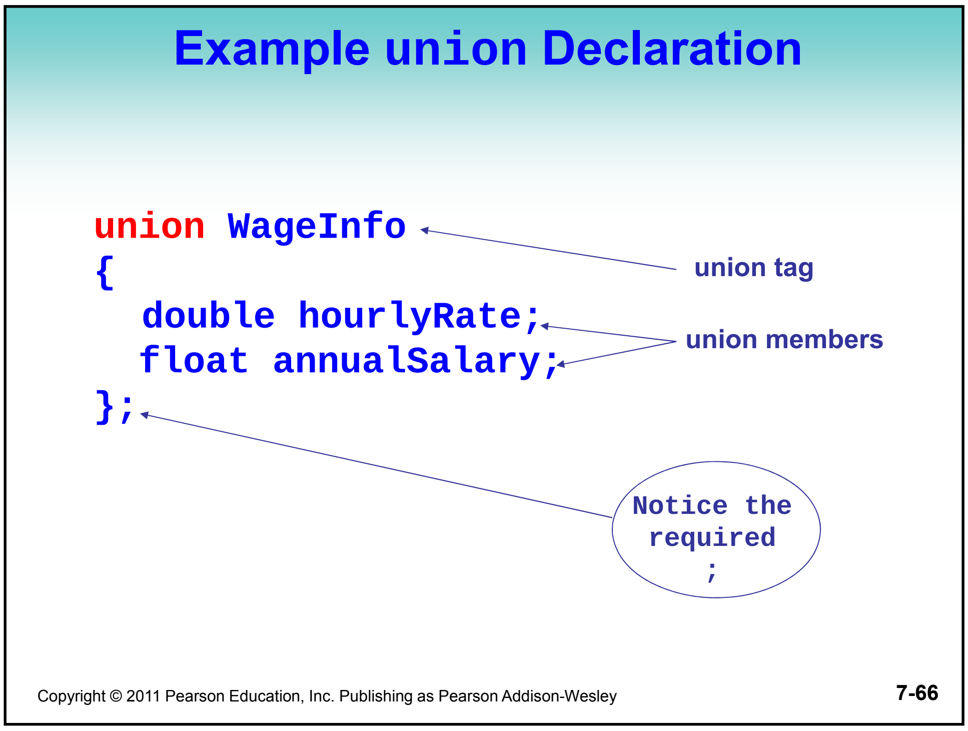
```
Student getStuData()
{
  Student s;    // local variable
  cin >> s.studentID;
  cin.ignore();
  getline(cin, s.pData.name);
  getline(cin, s.pData.address);
  getline(cin, s.pData.city);
  cin >> s.year;
  cin >> s.gpa;
  return s;
}
```



Unions

許多 **data fields** 或許可以共享同樣的記憶空間，以節省空間
➔ 使用 ***Union*** ， 但是 **data fields** 必須是互斥的，也就是說不會同時存在

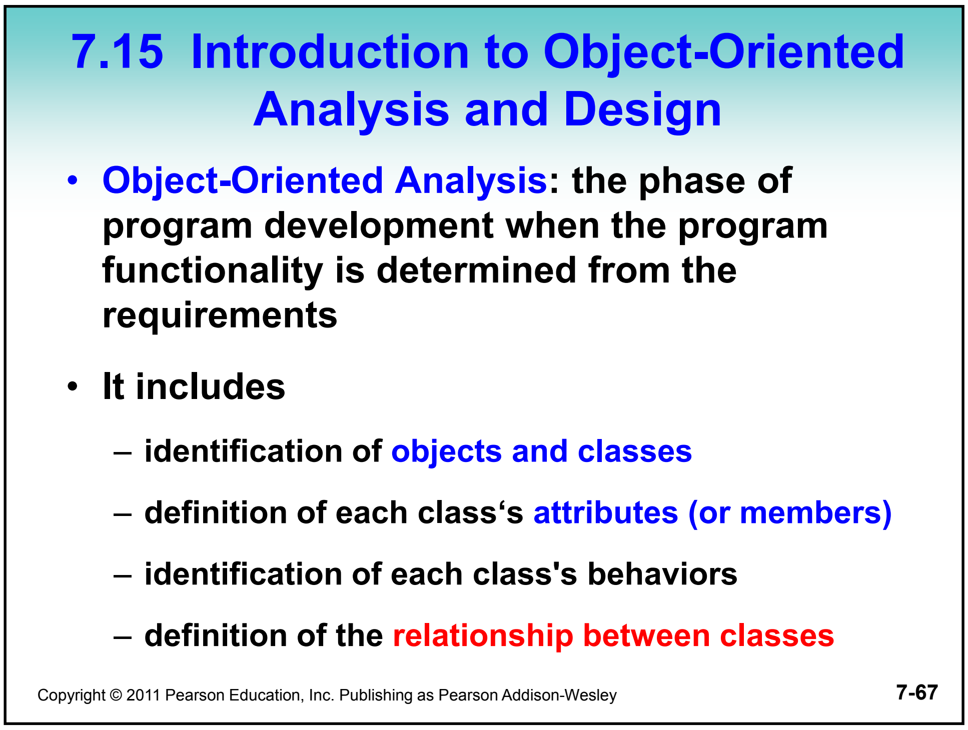
- Similar to a `struct`, but
  - all members in a "union structure" <u>share a single memory location</u>, thereby <u>saving space</u>
  - only 1 member of the union can be used at a time
- Declared using key word union
- Otherwise the same as `struct`
- Variables defined and accessed like `struct` variables



Example union Declaration

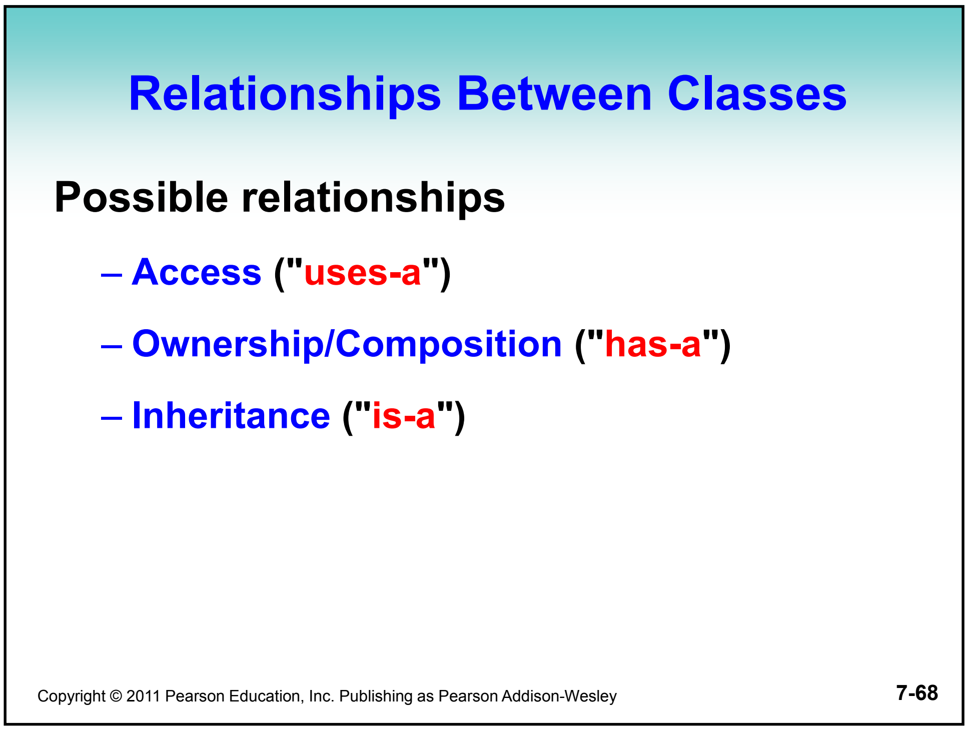
```
union WageInfo
{
   double hourlyRate;
   float annualSalary;
};
```

union tag
union members
Notice the required ;



7.15 Introduction to Object-Oriented Analysis and Design

- **Object-Oriented Analysis**: the phase of program development when the program functionality is determined from the requirements
- It includes
  - identification of objects and classes
  - definition of each class's attributes (or members)
  - identification of each class's behaviors
  - definition of the relationship between classes



Relationships Between Classes

Possible relationships

- **Access** ("uses-a")
- **Ownership/Composition** ("has-a")
- **Inheritance** ("is-a")

Object Reuse

Class 可以形成 Library ➔ 支援重複的使用
For example, STL (Standard Template Library)

- A well-defined class can be used to create objects in multiple programs
- By re-using an object definition, program development time is shortened
- One goal of object-oriented programming is to support object reuse

7.16 Screen Control

- Programs to date have all displayed output starting at the upper left corner of computer screen or output window. Output is displayed left-to-right, line-by-line.
- Computer operating systems are designed to allow programs to access any part of the computer screen. Such access is operating system-specific.

Screen Control – Concepts

- An output screen can be thought of as a grid of 25 rows and 80 columns. Row 0 is at the top of the screen. Column 0 is at the left edge of the screen.
- The intersection of a row and a column is a cell. It can display a single character.
- A cell is identified by its row and column number. These are its coordinates.

Screen Control – Windows – Specifics

- `#include <windows.h>` to access the operating system from a program
- Create a HANDLE to reference the output screen:

```
HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
```

- Create a COORD structure to hold the coordinates of a cell on the screen:

```
COORD position;
```

Screen Control – Windows – More Specifics

- Assign coordinates where the output should appear:

```
position.X = 30;    // column
position.Y = 12;    // row
```

- Set the screen cursor to this cell:

```
SetConsoleCursorPosition(screen, position);
```

- Send output to the screen:

```
cout << "Look at me!" << endl;
```

  – be sure to end with `endl`, not `'\n'` or nothing

Chapter 8: Arrays

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Addison-Wesley
is an imprint of
PEARSON

Topics

8.1 Arrays for Holding Multiple Values
8.2 Accessing Array Elements
8.3 Input and Display of Array Contents
8.4 Array Initialization
8.5 Process of Array Contents
8.6 Parallel Arrays

Topics (continued)

8.7 The `typedef` Statement

8.8 Arrays as Function Arguments

8.9 Two-Dimensional Arrays

8.10 Arrays with Three or More Dimensions

8.11 Vectors

8.12 Arrays of Class Objects

8.1 Arrays for Holding Multiple Values

- Array: variable that can store multiple values of the same type
- Values are stored in adjacent memory locations
- Declared using [] operator

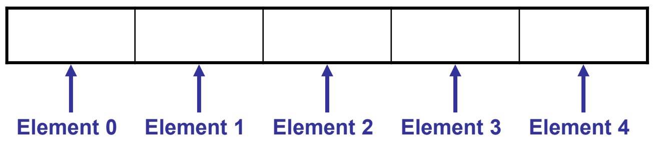
```
const int ISIZE = 5;
int tests[ISIZE];
```

Array Storage in Memory

The definition

```
int tests[ISIZE];  // ISIZE is 5
```

allocates the following memory



Array Terminology

In the definition `int tests[ISIZE];`

- `int` is the data type of the array elements
- `tests` is the name of the array
- `ISIZE`, in `[ISIZE]`, is the size declarator. It shows the number of elements in the array.
- The size of an array is the number of bytes allocated for it

*(number of elements) * (bytes needed for each element)*

Array Terminology Examples

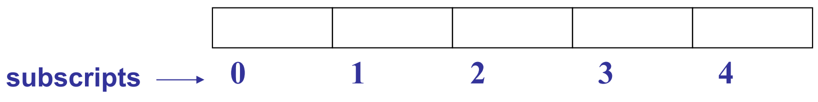
Examples:

Assumes `int` uses 4 bytes and `double` uses 8 bytes

```
 const int ISIZE = 5, DSIZE = 10;
 int tests[ISIZE]; // holds 5 ints, array
                   // occupies 20 bytes
double volumes[DSIZE];// holds 10 doubles
                      // array is 80 bytes
```

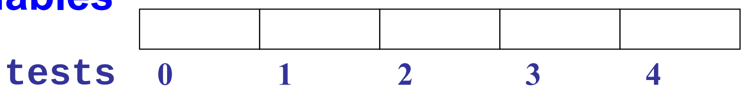
8.2 Accessing Array Elements

- Each array element has a subscript, used to access the element.
- Subscripts start at 0



Access of Array Elements

Array elements (accessed by array name and subscript) can be used **as regular variables**



```
tests[0] = 79;
cout << tests[0];
cin  >> tests[1];
tests[4] = tests[0] + tests[1];
cout << tests; // illegal due to
               // missing subscript
```

8.3 Input and Display of Array Contents

cout and cin can be used to display values from and store values into an array

```
const int ISIZE = 5;

int tests[ISIZE]; // Define 5-elt. array
cout << "Enter first test score ";
cin  >>  tests[0];
```

Array Subscripts

- Array subscript can be an integer constant, integer variable, or integer expression
- Examples:

	Subscript is
`cin  >> tests[3];`	int constant
`cout << tests[i];`	int variable
`cout << tests[i+j];`	int expression

Input and Display of All Array Elements

To access each element of an array

- Use a loop
- Let the loop control variable be the array subscript
- A different array element will be referenced each time through the loop

```
for (i = 0; i < 5; i++)
   cout << tests[i] << endl;
```

Getting Array Data from a File

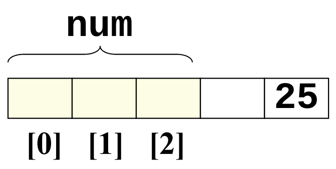
```
const int ISIZE = 5, sales[ISIZE];
ifstream dataFile;
datafile.open("sales.dat");
if (!dataFile)
   cout << "Error opening data file\n";
else
{  // Input daily sales
   for (int day = 0; day < ISIZE; day++)
      dataFile >> sales[day];
   dataFile.close();
}
```

No Bounds Checking

- There are no checks in C++ that an array subscript is in range?
- An invalid array subscript can cause program to overwrite other memory
- Example:

```
const int ISIZE = 3;
int i = 4;
int num[ISIZE];
num[i] = 25;
```

Out of Range Error!



Off-By-One Errors

- Most often occur when a program accesses data one position beyond the end of an array, or misses the first or last element of an array.
- Don’t confuse the ordinal number of an array element (first, second, third) with its subscript (0, 1, 2)

8.4 Array Initialization

- Can be initialized during program execution with assignment statements

```
tests[0] = 79;
tests[1] = 82; // etc.
```

- Can be initialized at array definition with an initialization list

```
const int ISIZE = 5;
int tests[ISIZE] = {79,82,91,77,84};
```

Start at element 0 or 1?

- May choose to declare arrays to be one larger than needed. This allows you to use the element with subscript 1 as the 'first' element, etc., and may minimize off-by-one errors. 策略之一: 不用註標為0的位置
- **Element with subscript 0 is not used!**
- This is most often done when working with ordered data, *e.g.*, months of the year or days of the week

Partial Array Initialization

- If array is initialized at definition with fewer values than the size declarator of the array, remaining elements will be set to `0` or `NULL`

```
int tests[ISIZE] = {79, 82};
```

79	82	0	0	0

- Initial values used in order; cannot skip over elements to initialize noncontiguous range

Implicit Array Sizing

- Can determine array size by the size of the initialization list

```
short quizzes[]={12,17,15,11};
```

12	17	15	11

- Must use either array size declarator or initialization list when array is defined

8.5 Processing Array Contents

- Array elements can be
  - treated as ordinary variables of the same type as the array
  - used in arithmetic operations, in relational expressions, etc.
- Example:

```
if (principalAmt[3] >= 10000)
  interest = principalAmt[3] * intRate1;
else
  interest = principalAmt[3] * intRate2;
```

Using Increment and Decrement Operators with Array Elements

When using ++ and - - operators, don't confuse the element with the subscript

```
tests[i]++;  // adds 1 to tests[i]
tests[i++];  // increments i, but has
             // no effect on tests
```

Copying One Array to Another

- Cannot copy with an assignment statement:

```
tests2 = tests;  //won't work
```

- Must instead use a loop to copy element-by-element:

```
for (int indx=0; indx < ISIZE; indx++)
   tests2[indx] = tests[indx];
```

Are Two Arrays Equal?

- Like copying, cannot compare in a single expression:

```
if (tests2 == tests)  //won't work
```

- Use a while loop with a boolean variable:

```
bool areEqual=true; // a default result
int indx=0;
while (areEqual && indx < ISIZE)
{
   if(tests[indx] != tests2[indx]
      areEqual = false; index++;
}
```

Sum, Average of Array Elements

- Use a simple loop to add together array elements

```
float average, sum = 0;
for (int tnum=0; tnum< ISIZE; tnum++)
   sum += tests[tnum];
```

- Once summed, average can be computed

```
average = sum/ISIZE;
```

Largest Array Element

- Use a loop to examine each element and find the largest element (*i.e.*, one with the largest value)

```
int largest = tests[0];
for (int tnum = 1; tnum < ISIZE; tnum++)
{  if (tests[tnum] > largest)
        largest = tests[tnum];
}
cout << "Highest score is " << largest;
```

- A similar algorithm exists to find the smallest element

Partially-Filled Arrays

- The exact amount of data (and, therefore, array size) may not be known when a program is written.
- Programmer makes best estimate for maximum amount of data, sizes arrays accordingly. A sentinel value can be used to indicate end-of-data. 實際有效的最後一筆 資料位置
- Programmer must also keep track of how many array elements are actually used

C-Strings and `string` Objects

Can be processed using array name

- Entire string at once, or
- One element at a time by using a subscript

```
string city;
cout << "Enter city name: ";
cin  >> city;
```

'S'	'a'	'l'	'e'	'm'
city[0]	city[1]	city[2]	city[3]	city[4]

8.7 The typedef Statement

- Creates an alias for a simple or structured data type
- Format:

```
typedef existingType newName;
```

- Example:

```
typedef unsigned int Uint;
Uint tests[ISIZE]; // array of
                   // unsigned ints
```

Uses of typedef

- Used to make code more readable
- Can be used to create alias for array of a particular type

```
// Define yearArray as a data type
// that is an array of 12 ints
typedef int yearArray[MONTHS];

// Create two of these arrays
yearArray highTemps, lowTemps;
```

8.8 Arrays as Function Arguments

- To define a function that has an array parameter, use empty [] to indicate the array argument
- To pass an array to a function, just use the array name

```
// Function prototype
void showScores(int []);

// Function header
void showScores(int tests[])

// Function call
showScores(tests);
```

Passing an Array Element

- Passing a single array element to a function is no different than passing a regular variable of that data type
- Function does not need to know that the value it receives is coming from an array

```
displayValue(score[i]);     // call
void displayValue(int item) // header
{  cout << item << endl;
}
```

Passing an Entire Array

- Use the array name, without any brackets, as the argument
- Can also pass the array size so the function knows how many elements to process

```
showScores(tests, 5);        // call
void showScores(int[], int); // prototype
void showScores(int A[],
                int size) // header
```

Using typedef with a Passed Array

Can use `typedef` to simplify function prototype and heading

```
// Make intArray an integer array
// of unspecified size
typedef int intArray[];

// Function prototype
void showScores(intArray, int);

// Function header
void showScores(intArray tests,
                         int size)
```

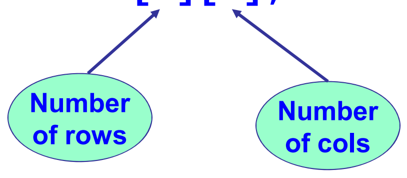
Modifying Arrays in Functions

- Array parameters in functions are similar to reference variables
- Changes made to array in a function are made to the actual array in the calling function
- Must be careful that an array is not inadvertently changed by a function!

8.9 Two-Dimensional Arrays

- Can define one array for multiple sets of data
- Like a table in a spreadsheet
- Use two size declarators in definition

```
int exams[4][3];
```



Two-Dimensional Array Representation

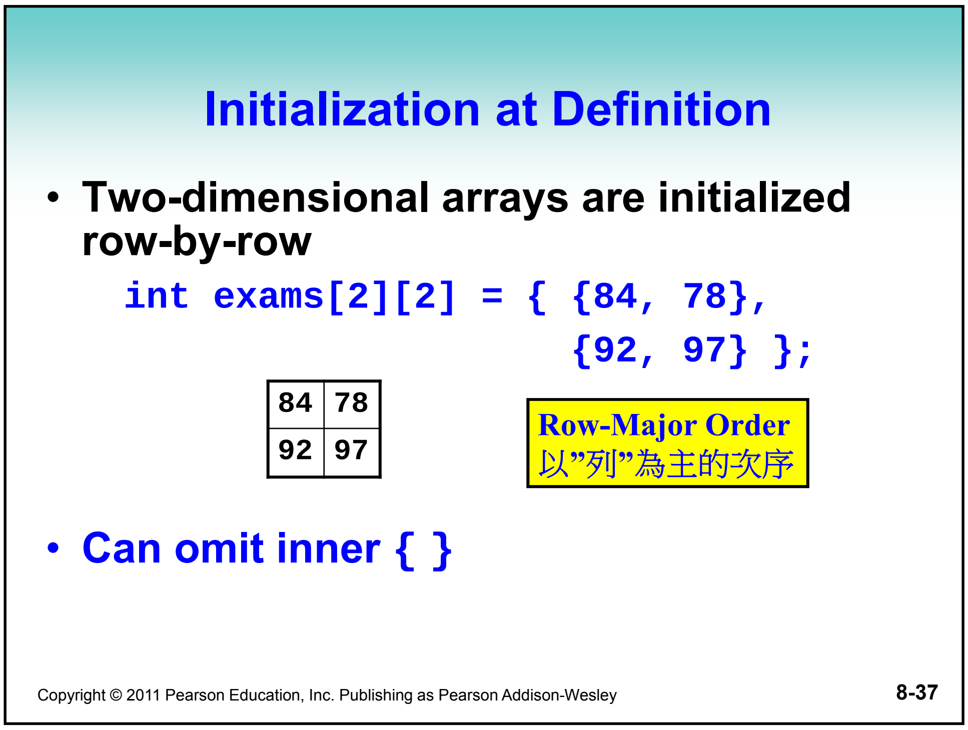
```
int exams[4][3];
```

columns

rows			
r	exams[0][0]	exams[0][1]	exams[0][2]
o	exams[1][0]	exams[1][1]	exams[1][2]
w	exams[2][0]	exams[2][1]	exams[2][2]
s	exams[3][0]	exams[3][1]	exams[3][2]

Use two subscripts to access element

```
exams[2][2] = 86;
```



Initialization at Definition

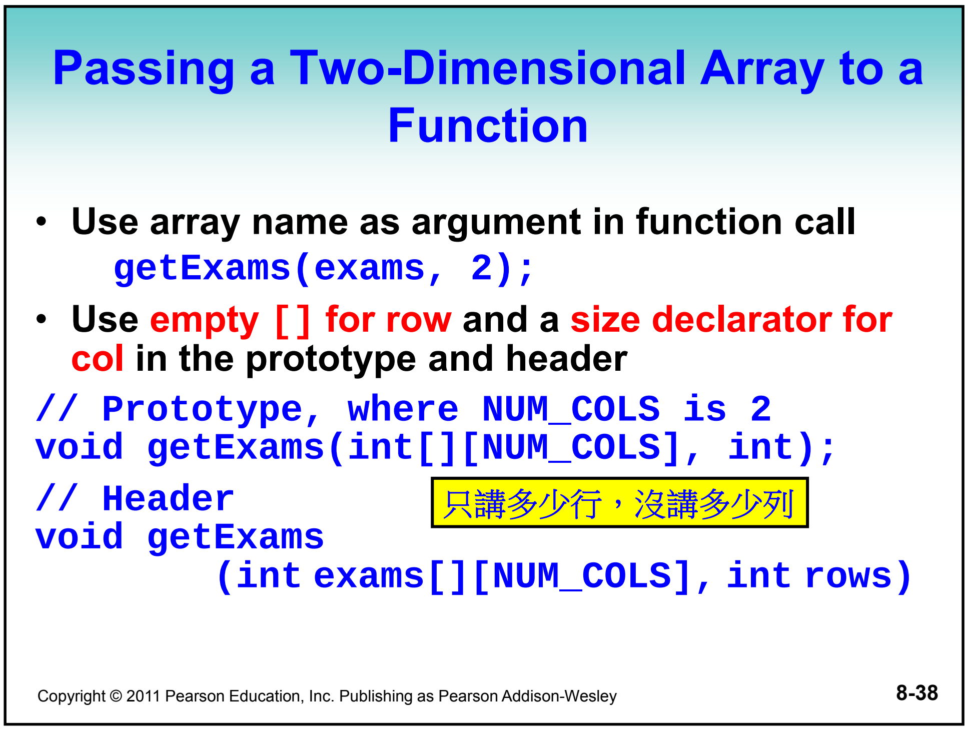
- Two-dimensional arrays are initialized row-by-row

```
int exams[2][2] = { {84, 78},
                    {92, 97} };
```

84	78
92	97

Row-Major Order
以”列”為主的次序

- Can omit inner { }



Passing a Two-Dimensional Array to a Function

- Use array name as argument in function call

```
getExams(exams, 2);
```

- Use empty [] for row and a size declarator for col in the prototype and header

```
// Prototype, where NUM_COLS is 2
void getExams(int[][NUM_COLS], int);
// Header
void getExams
        (int exams[][NUM_COLS], int rows)
```

只講多少行，沒講多少列

Using typedef with a Two-Dimensional Array

Can use `typedef` for simpler notation

```
typedef int intExams[][2];
  ...
// Function prototype
void getExams(intExams, int);
// Function header
void getExams(intExams exams, int rows)
```

直接訂一個 2-維陣列的 type

2D Array Traversal

- Use nested loops, one for row and one for column, to visit each array element.
- Accumulators can be used to sum the elements row-by-row, column-by-column, or over the entire array.

8.10 Arrays with Three or More Dimensions

- Can define arrays with any number of dimensions

```
short rectSolid(2,3,5);
double timeGrid(3,4,3,4);
```

- When used as parameter, specify size of all but 1st dimension

```
void getRectSolid(short [][3][5]);
```

STL中最常用的 Class

8.11 Vectors

- Holds a set of elements, like an array
- Flexible number of elements - can grow and shrink 具伸縮性
  - No need to specify size when defined
  - Automatically adds more space as needed
- Defined in the Standard Template Library (STL)
  - Covered in a later chapter
- Must include `vector` header file to use vectors

```
#include <vector>
```

Vectors

- Can hold values of any type
  - Type is specified when a vector is defined

```
vector<int> scores;
vector<double> volumes;
```

要宣告 element 是甚麼 type

- Can use [] to access elements

如同內建的 Array 一般

Defining Vectors

- Define a vector of integers (starts with 0 elements)

```
vector<int> scores;
```

- Define int vector with initial size 30 elements

```
vector<int> scores(30);
```

- Define 20-element int vector and initialize all elements to 0

```
vector<int> scores(20, 0);
```

2nd 參數是起始值!

- Define int vector initialized to size and contents of vector finals

```
vector<int> scores(finals);
```

Growing a Vector's Size

- Use push_back member function to add an element to a full array or to an array that had no defined size

```
// Add a new element holding a 75
scores.push_back(75);
```

- Use size() member function to determine number of elements currently in a vector

```
howbig = scores.size();
```

Vector 的 size 就是element 個數

Removing Vector Elements

- Use pop_back member function to remove last element from vector

```
scores.pop_back();
```

- To remove all contents of vector, use clear member function

```
scores.clear();
```

- To determine if vector is empty, use empty member function

```
while (!scores.empty()) ...
```

8.14 Arrays of Class Objects

- Class objects can also be used as array elements

```
class Square
{ private:
    int side;
  public:
    Square(int s = 1)   Constructor
    { side = s; }
    int getSide()
    { return side; }
};
Square shapes[10];  // Create array of 10
                    // Square objects
```

Arrays of Class Objects

- Like an array of structures, use an array subscript to access a specific object in the array
- Then use dot operator to access member methods of that object

Member Methods
就是 Member Functions

```
for (i = 0; i < 10; i++)
  cout << shapes[i].getSide() << endl;
```

Initializing Arrays of Objects

- Can use default constructor to perform same initialization for all objects
- Can use initialization list to supply specific initial values for each object

```
Square shapes[5] = {1,2,3,4,5};
```

- Default constructor is used for the remaining objects if initialization list is too short

```
Square boxes[5] = {1,2,3};
```

Initializing Arrays of Objects

If an object is initialized with a constructor that takes > 1 argument, the initialization list must include a call to the constructor for that object

```
 Rectangle spaces[3] =
{ Rectangle(2,5),
  Rectangle(1,3),
  Rectangle(7,7)  };
```

如果某個物件的起始
需要超過一個以上的參數，
則必須明白的寫出 constructor

8.13 Arrays of Structures

- Structures can be used as array elements

```
 struct Student
{
   int studentID;
   string name;
   short year;
   double gpa; // grade point average
};
const int CSIZE = 30;
Student class[CSIZE]; // Holds 30
                      // Student structures
```

Arrays of Structures

- Use array **subscript** to access a specific structure in the array
- Then use **dot operator** to access members of that structure

```
cin  >> class[25].studentID;
cout << class[i].name << " has GPA "
     << class[i].gpa << endl;
```

Chapter 9: Searching, Sorting, and Algorithm Analysis

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

Topics

- 9.1 Introduction to Search Algorithms
- 9.2 Searching an Array of Objects
- 9.3 Introduction to Sorting Algorithms
- 9.4 Sorting an Array of Objects
- 9.5 Sorting and Searching Vectors
- 9.6 Introduction to Analysis of Algorithms

9.1 Introduction to Search Algorithms

- **Search**: locate an item in a list (array, vector, etc.) of information
- Two algorithms (methods) considered here:
  - Linear search
  - Binary search

Linear Search Algorithm

```
Set found to false
Set position to –1
Set index to 0
While index < number of elts and found is false
   If list [index] is equal to search value
      found = true
      position = index
   End If
   Add 1 to index
End While
Return position
```

Linear Search Example

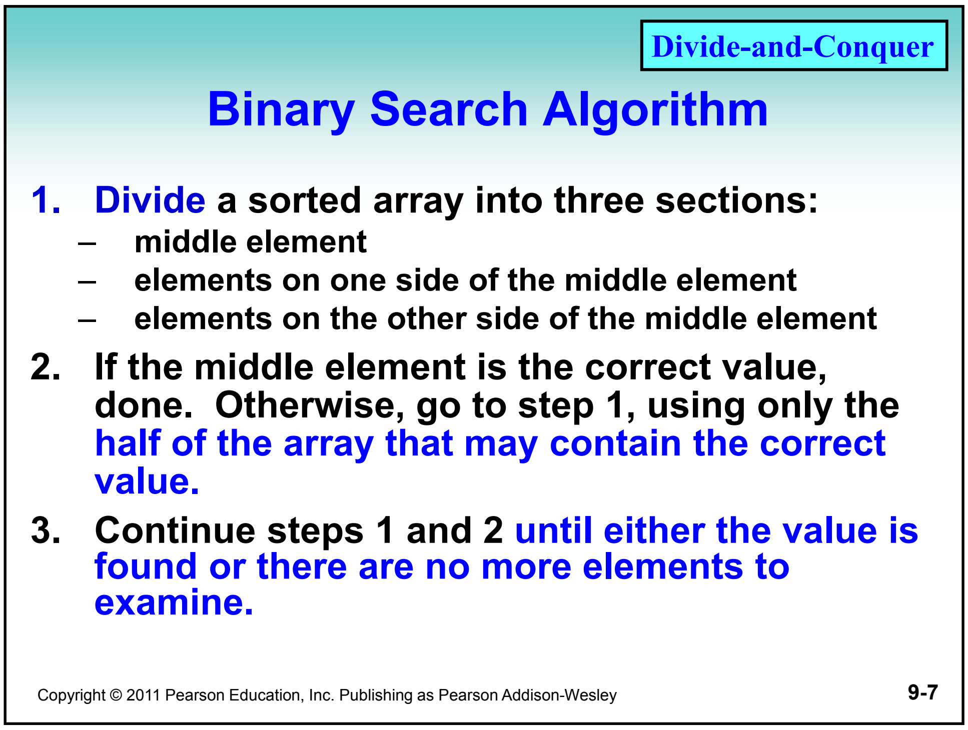
- Array `numlist` contains

17	23	5	11	2	29	3

- Searching for the value 11, linear search examines `17, 23, 5,` and `11`
- Searching for the value 7, linear search examines `17, 23, 5, 11, 2, 29,` and `3`

Linear Search Tradeoffs

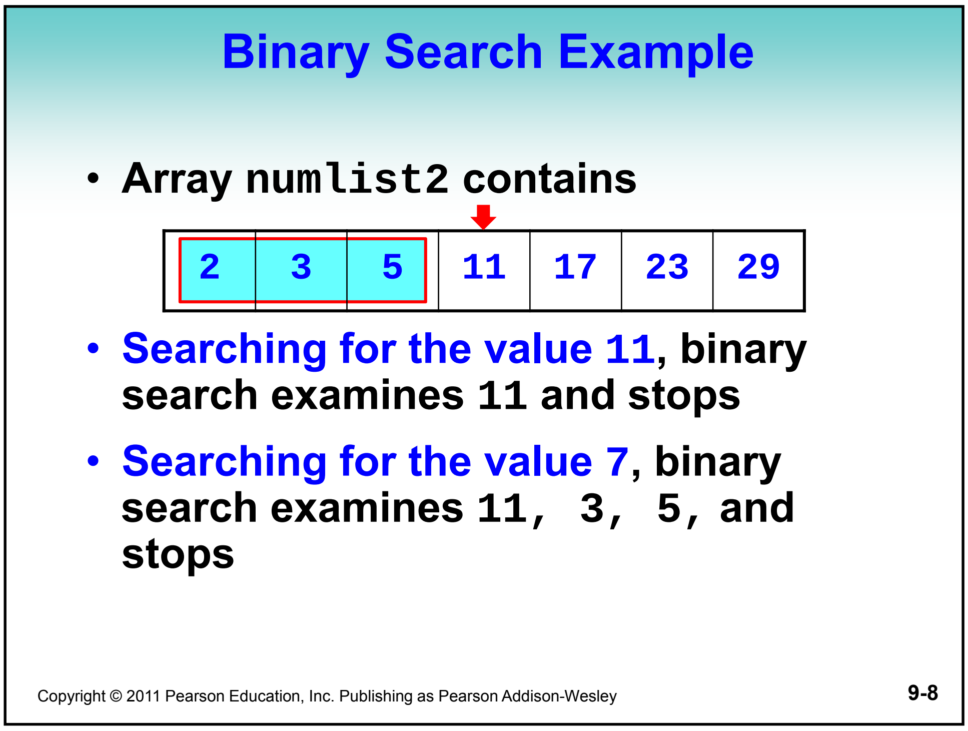
- Benefits
  - Easy algorithm to understand
  - Array can be in any order
- Disadvantage
  - Inefficient (slow): for array of N elements, examines N/2 elements on average for value that is found in the array, N elements for value that is not in the array



Divide-and-Conquer

Binary Search Algorithm

1. Divide a sorted array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1, using only the half of the array that may contain the correct value.
3. Continue steps 1 and 2 until either the value is found or there are no more elements to examine.

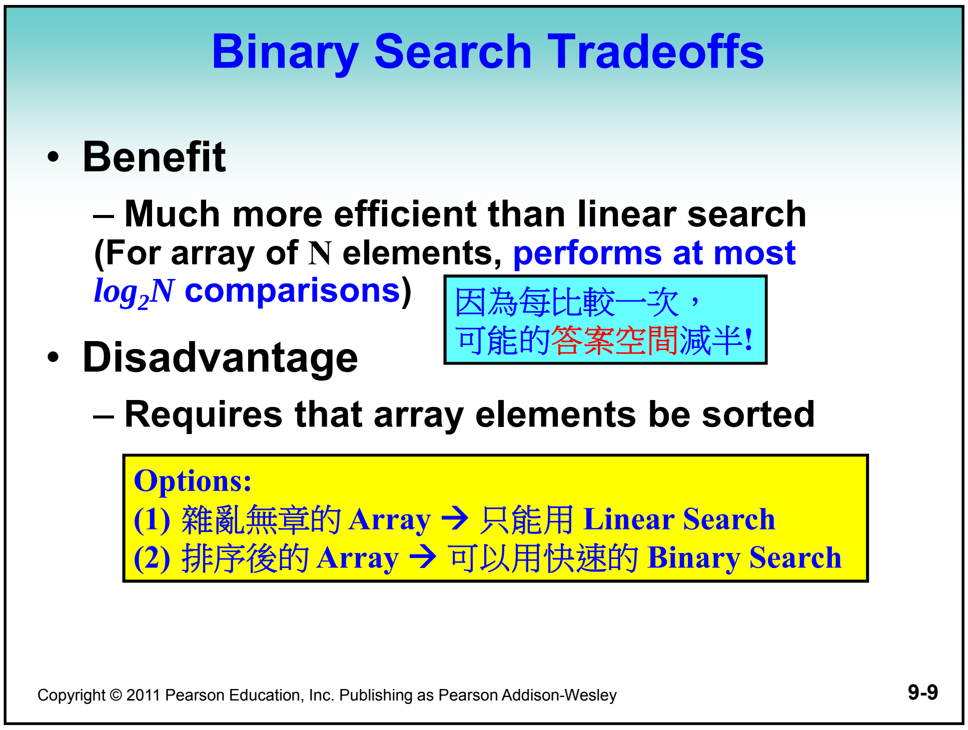


Binary Search Example

- Array `numlist2` contains

2	3	5	11	17	23	29

- Searching for the value 11, binary search examines 11 and stops
- Searching for the value 7, binary search examines `11, 3, 5,` and stops



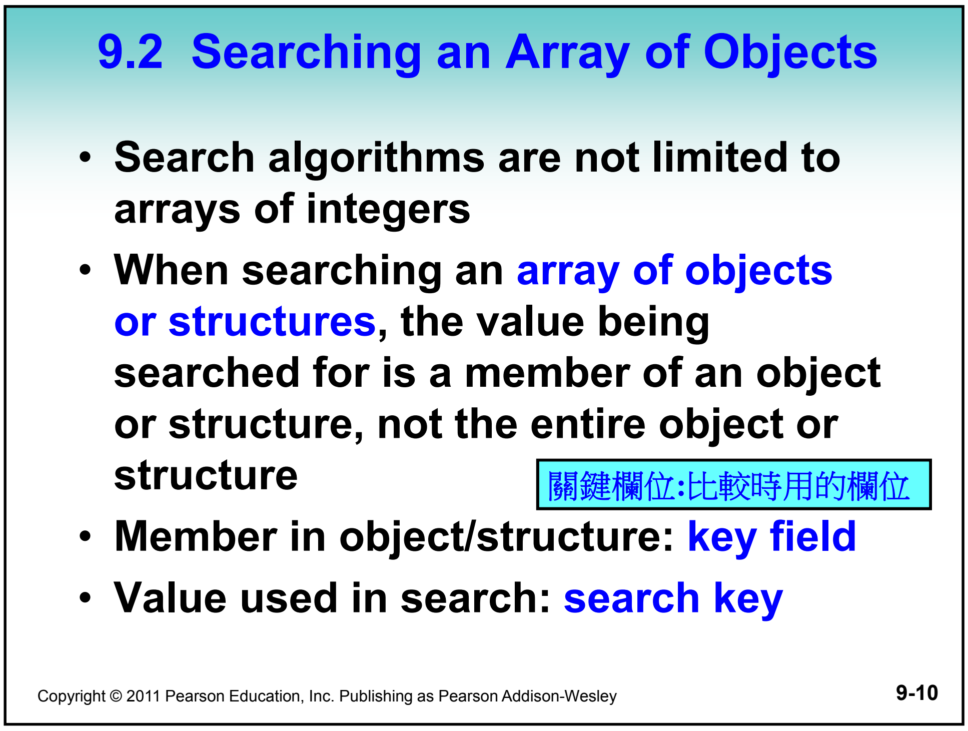
Binary Search Tradeoffs

- Benefit
  - Much more efficient than linear search (For array of N elements, performs at most log_2N comparisons)

因為每比較一次，
可能的答案空間減半!

- Disadvantage
  - Requires that array elements be sorted

Options:
(1) 雜亂無章的 Array → 只能用 Linear Search
(2) 排序後的 Array → 可以用快速的 Binary Search



9.2 Searching an Array of Objects

- Search algorithms are not limited to arrays of integers
- When searching an array of objects or structures, the value being searched for is a member of an object or structure, not the entire object or structure

關鍵欄位:比較時用的欄位

- Member in object/structure: key field
- Value used in search: search key

9.3 Introduction to Sorting Algorithms

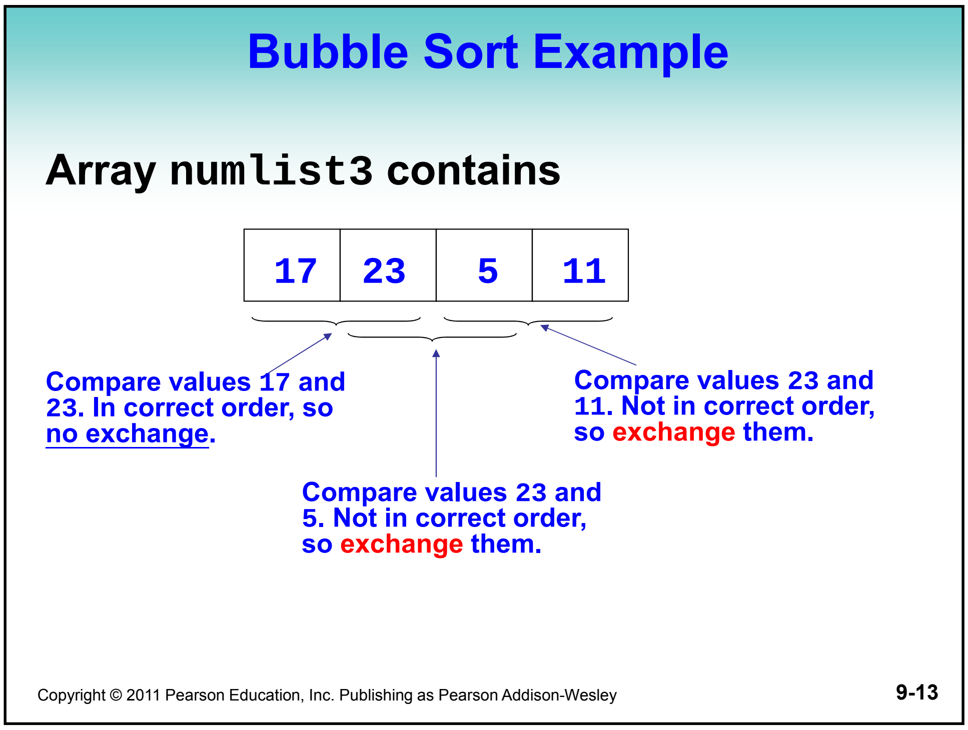
- Sort: arrange values into an order
  - Alphabetical
  - Ascending numeric
  - Descending numeric
- Two algorithms considered here
  - Bubble sort
  - Selection sort

(假設要排成 Increasing order)➔哪就讓重的泡泡有次序的往下沉…

Bubble Sort Algorithm

1. Compare 1st two elements and exchange them if they are out of order. Basic Operation
2. Move down one element and compare 2nd and 3rd elements. Exchange if necessary. Continue until end of array.
3. Pass through array again, repeating process and exchanging as necessary.
4. Repeat until a pass is made with no exchanges.

終止條件



Bubble Sort Example

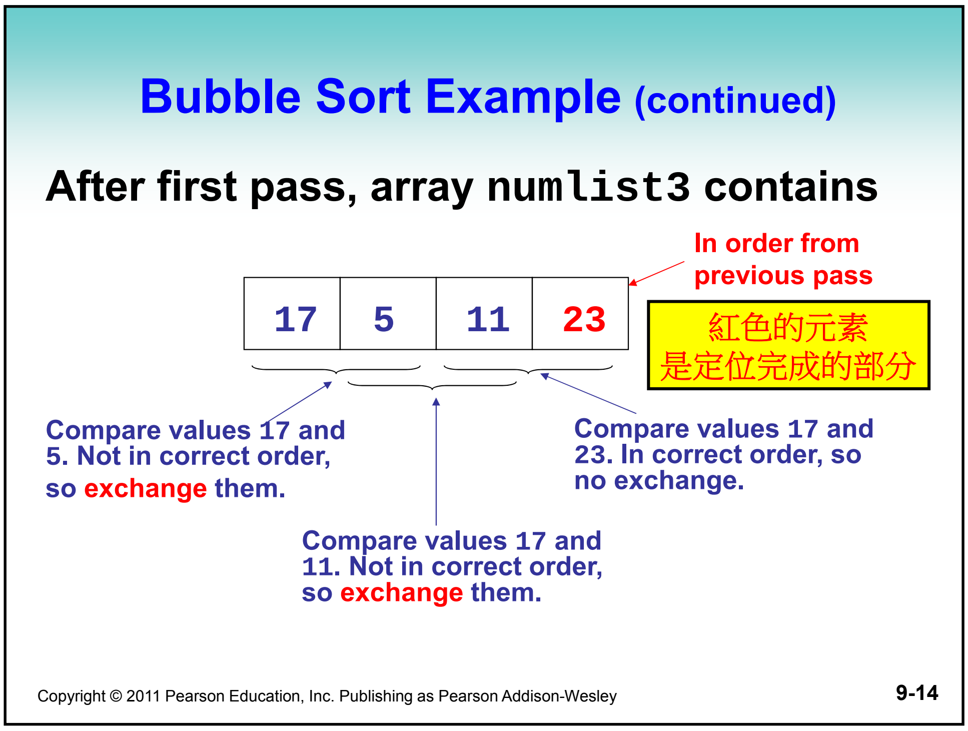
Array `numlist3` contains

17	23	5	11

Compare values 17 and 23. In correct order, so no exchange.

Compare values 23 and 5. Not in correct order, so exchange them.

Compare values 23 and 11. Not in correct order, so exchange them.



Bubble Sort Example (continued)

After first pass, array `numlist3` contains

17	5	11	23

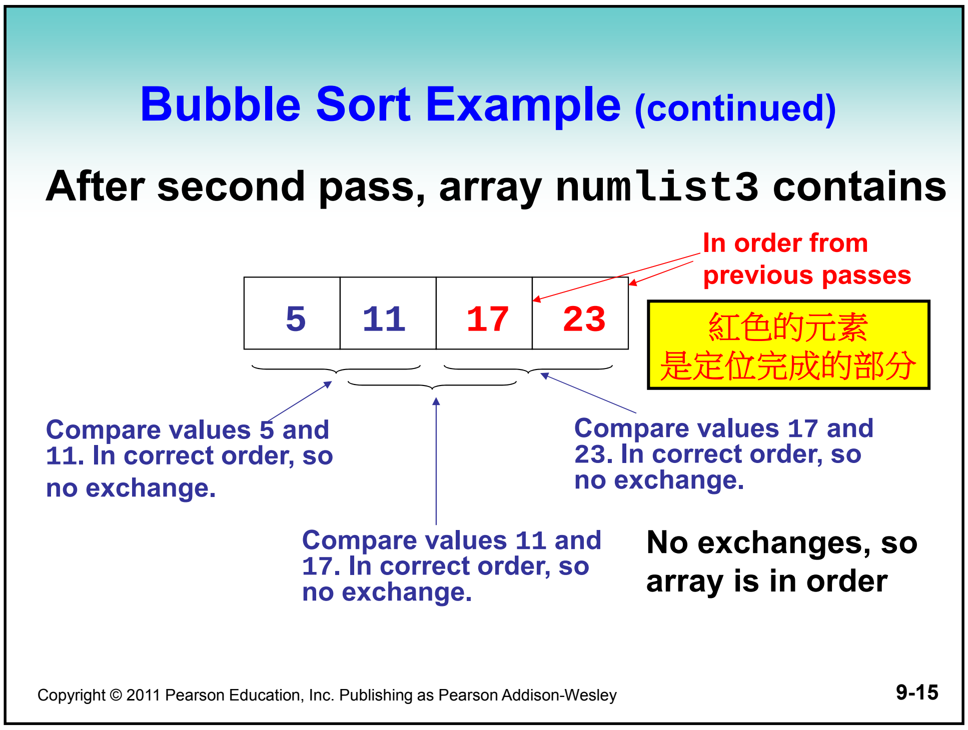
In order from previous pass

紅色的元素
是定位完成的部分

Compare values 17 and 5. Not in correct order, so exchange them.

Compare values 17 and 11. Not in correct order, so exchange them.

Compare values 17 and 23. In correct order, so no exchange.



Bubble Sort Example (continued)

After second pass, array `numlist3` contains

In order from previous passes

5	11	17	23

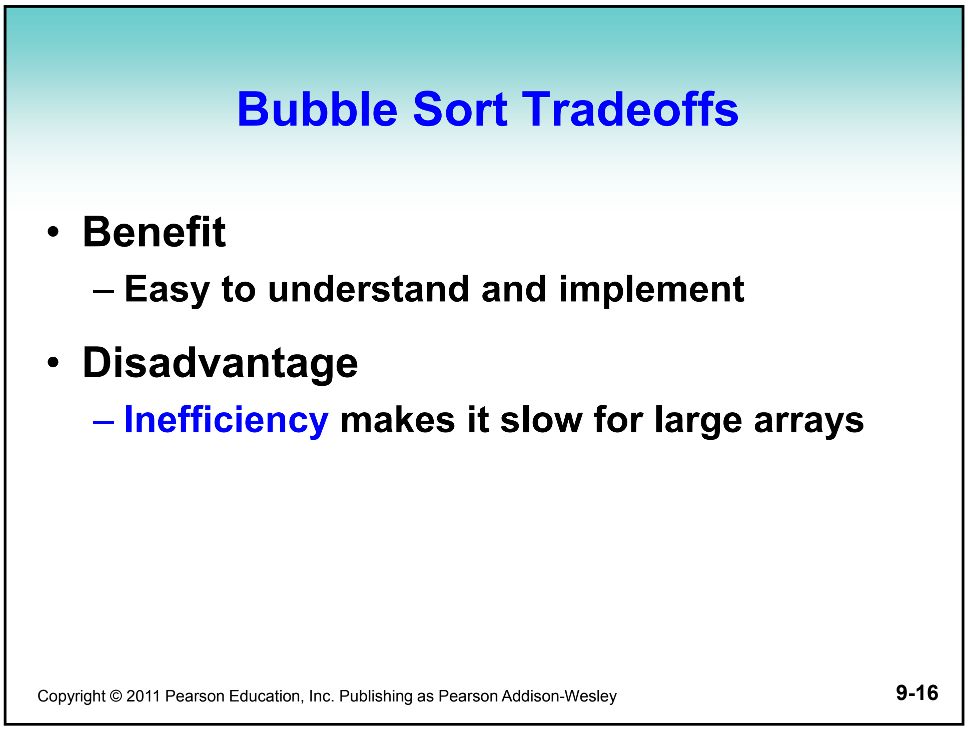
紅色的元素
是定位完成的部分

Compare values 5 and 11. In correct order, so no exchange.

Compare values 11 and 17. In correct order, so no exchange.

Compare values 17 and 23. In correct order, so no exchange.

No exchanges, so array is in order



Bubble Sort Tradeoffs

- Benefit
  - Easy to understand and implement
- Disadvantage
  - Inefficiency makes it slow for large arrays

Selection Sort Algorithm

1. **Locate smallest element** in array and exchange it with element in position 0.
2. **Locate next smallest element** in array and exchange it with element in position 1.
3. **Continue** until all elements are in order.

Selection Sort Example

Array `numlist` contains

11	2	29	3

Smallest element is 2. Exchange 2 with element in 1st array position (*i.e.* element 0).

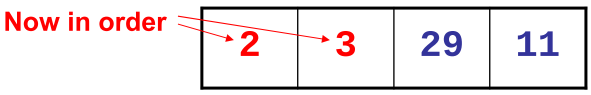
Now in order

紅色的元素
是定位完成的部分

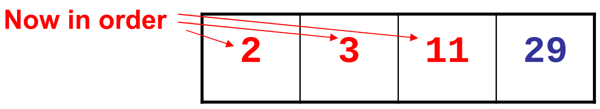
2	11	29	3

Selection Sort – Example (continued)

Next smallest element is 3. Exchange 3 with element in 2nd array position.



Next smallest element is 11. Exchange 11 with element in 3rd array position.



Selection Sort Tradeoffs

- Benefit
  - More efficient than Bubble Sort, due to fewer exchanges
- Disadvantage
  - Considered harder than Bubble Sort to understand (但是這其實不能算是甚麼真的缺點!)

9.4 Sorting an Array of Objects

- As with searching, arrays to be sorted can contain objects or structures
- The key field determines how the structures or objects will be ordered
- When exchanging contents of two array elements, entire structures or objects must be exchanged, not just the key fields in the structures or objects

9.5 Sorting and Searching Vectors

- Sorting and searching algorithms can be applied to vectors as well as to arrays
- Need slight modifications to functions to use vector arguments
  - `vector <type> &` used in prototype
  - No need to indicate vector size as functions can use `size` member function to calculate

9.6 Introduction to Analysis of Algorithms

- (重要問題) Given two algorithms to solve a problem, what makes one better than the other?
- Efficiency of an algorithm is measured by
  - space (computer memory used)
  - time (how long to execute the algorithm)
- Analysis of algorithms is a more effective way to decide efficiency than by using empirical data

Analysis of Algorithms: Terminology

- Computational Problem: problem solved by an algorithm
- Basic step: operation in the algorithm that executes in a constant amount of time
- Examples of basic steps:
  - exchange the contents of two variables
  - compare two values

Analysis of Algorithms: Terminology

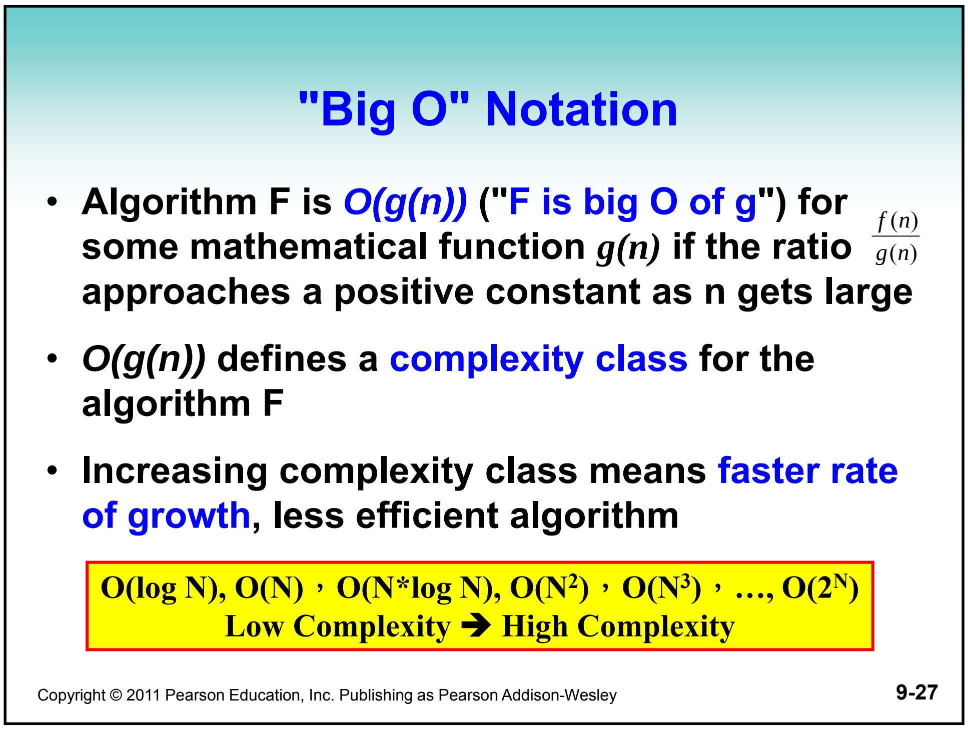
- Complexity of an algorithm: the number of basic steps required to execute the algorithm depends on the input size N (N input values)
- Worst-case complexity of an algorithm: number of basic steps for input size N in the worst case
- Average case complexity function: number of basic steps for input size N in the average

Comparison of Algorithmic Complexity

Asymptotic Complexity
漸近線複雜度
➔ 當input size 很大時

Given algorithms F and G with complexity functions *f(n)* and *g(n)* for input of size n

- If the ratio $\frac{f(n)}{g(n)}$ approaches a constant value as n gets large, F and G have equivalent efficiency
- If the ratio $\frac{f(n)}{g(n)}$ gets larger as n gets large, algorithm G is more efficient than algorithm F
- If the ratio $\frac{f(n)}{g(n)}$ approaches 0 as n gets large, algorithm F is more efficient than algorithm G



"Big O" Notation

- Algorithm F is *O(g(n))* ("F is big O of g") for some mathematical function *g(n)* if the ratio $\frac{f(n)}{g(n)}$ approaches a positive constant as n gets large
- *O(g(n))* defines a complexity class for the algorithm F
- Increasing complexity class means faster rate of growth, less efficient algorithm

$O(\log N)$, $O(N)$，$O(N*\log N)$, $O(N^2)$，$O(N^3)$，…, $O(2^N)$
Low Complexity ➔ High Complexity

Chapter 10: Pointers

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Addison-Wesley is an imprint of PEARSON

Topics

10.1 Pointers and the Address Operator

10.2 Pointer Variables

10.3 The Relationship Between Arrays and Pointers

10.4 Pointer Arithmetic

10.5 Initializing Pointers

10.6 Comparing Pointers

Topics (continued)

10.7 Pointers as Function Parameters

10.8 Pointers to Constants and Constant Pointers

10.9 Dynamic Memory Allocation

10.10 Returning Pointers from Functions

10.11 Pointers to Class Objects and Structures

10.12 Selecting Members of Objects

10.1 Pointers and the Address Operator

- Each variable in a program is stored at a unique address in memory
- Use the address operator & to get the address of a variable:

```
int num = -23;
cout << &num; // prints address
              // in hexadecimal
```

- The address of a memory location is a pointer

10.2 Pointer Variables

- **Pointer variable** (**pointer**): variable that holds an address
- Pointers provide an alternate way to access memory locations

Pointer Variables

- Definition:

```
int  *intptr;
```

- Read as:
  "`intptr` can hold the address of an int"
  or "the variable that `intptr` points to has type int"
- Spacing in definition does not matter:

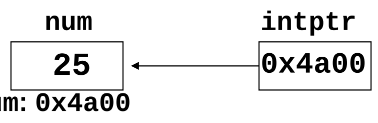
```
int * intptr;
int*  intptr;
```

Pointer Variables

- Assignment:

```
int num = 25;
int *intptr;
intptr = &num; // 左右兩邊 type 一樣
```

- Memory layout:



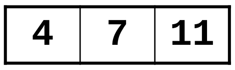
- Can access num using intptr and indirection operator *:

```
cout << intptr;  // prints 0x4a00
cout << *intptr; // prints 25
```

10.3 The Relationship Between Arrays and Pointers

- Array name is starting address of array

```
int vals[] = {4, 7, 11};
```



starting address of vals: 0x4a00

```
cout << vals;    // displays 0x4a00
cout << vals[0]; // displays 4
```

The Relationship Between Arrays and Pointers

- Array name can be used as a pointer constant

```
int vals[] = {4, 7, 11};
cout << *vals;    // displays 4
```

- Pointer can be used as an array name

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

Pointers in Expressions

- Given:

```
int vals[]={4,7,11};
int *valptr = vals;
```

- What is `valptr + 1?`
- It means (address in `valptr`) + (1 * size of an `int`)

```
cout << *(valptr+1); // displays 7
cout << *(valptr+2); // displays 11
```

- Must use ( ) in expression

Array Access

Array elements can be accessed in many ways

`int vals[]={4,7,11};`

Array access method	Example
array name and [ ]	`vals[2] = 17;`
pointer to array and [ ]	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals+2) = 17;`
pointer to array and subscript arithmetic	`*(valptr+2) = 17;`

Array Access

- Array notation

  `vals[i]`

  is equivalent to the pointer notation

  `*(vals + i)`

- No bounds checking performed on array access

10.4 Pointer Arithmetic

Some arithmetic operators can be used with pointers:

- Increment and decrement operators ++, - -
- Integers can be added to or subtracted from pointers using the operators +, -, +=, and -=
- One pointer can be subtracted from another by using the subtraction operator -

Pointer Arithmetic

Assume the variable definitions

```
int vals[]={4,7,11};
int *valptr = vals;
```

Examples of use of ++ and - -

```
valptr++; // points at 7
valptr--; // now points at 4
```

More on Pointer Arithmetic

Assume the variable definitions:

```
int vals[]={4,7,11};
int *valptr = vals;
```

Example of the use of + to add an int to a pointer:

```
cout << *(valptr + 2)
```

This statement will print 11

More on Pointer Arithmetic

Assume the variable definitions:

```
int vals[]={4,7,11};
int *valptr = vals;
```

Example of use of +=:

```
valptr = vals; // points at 4
valptr += 2;   // points at 11
```

More on Pointer Arithmetic

Assume the variable definitions

```
int vals[] = {4,7,11};
int *valptr = vals;
```

Example of pointer subtraction

```
valptr += 2;
cout << valptr - val;
```

This statement prints 2: the number of `int` between `valptr` and `val`

10.5 Initializing Pointers

- Can initialize to NULL or 0 (zero)

```
int *ptr = NULL;
```

- Can initialize to addresses of other variables

```
int num, *numPtr = &num;
int val[ISIZE], *valptr = val;
```

- Initial value must have correct type

```
float cost;
int *ptr = &cost; // won't work
```

左右 type 不合

10.6 Comparing Pointers

- Relational operators can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses
if (*ptr1 == *ptr2)  // compares
                     // contents
```

10.7 Pointers as Function Parameters

- A pointer can be a parameter
- Works like a reference parameter to allow change to argument from within function
- A pointer parameter must be explicitly de-referenced to access the contents at that address

```
*ptr_A
// De-reference 就是取內容之意
```

Pointers as Function Parameters

Requires:

(1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr);
```
副程式

(2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```
副程式

(3) address as argument to the function

```
getNum(&num);
```
主程式

Pointers as Function Parameters

副程式

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

主程式

```
int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

10.8 Ponters to Constants and Constant Pointers

- **Pointer to a constant**: cannot change the value that is pointed at
- **Constant pointer**: address in pointer cannot change once pointer is initialized

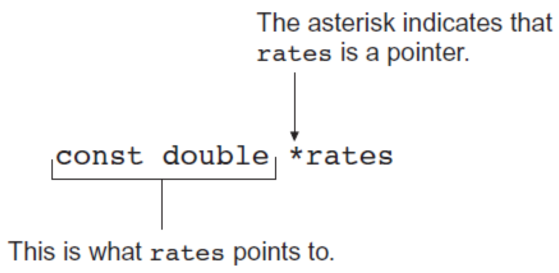
Pointers to Constant

- Must use `const` keyword in pointer definition:

```
const double taxRates[] =
                {0.65, 0.8, 0.75};
const double *ratePtr;
```

- Use `const` keyword for pointers in function headers to protect data from modification from within function

Pointer to Constant – What does the Definition Mean?


The asterisk indicates that rates is a pointer.
const double *rates
This is what rates points to.

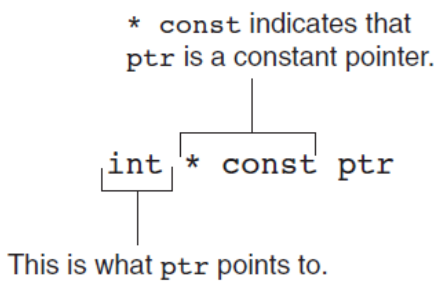
Constant Pointers

- Defined with `const` keyword adjacent to variable name:

```
int classSize = 24;
int * const classPtr = &classSize;
```

- Must be initialized when defined
- Can be used without initialization as a function parameter
  - Initialized by argument when function is called
  - Function can receive different arguments on different calls
- While the address in the pointer cannot change, the data at that address may be changed

Constant Pointer – What does the Definition Mean?


* const indicates that ptr is a constant pointer.
int * const ptr
This is what ptr points to.

10.9 Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Uses new operator to allocate memory

```
double *dptr;
dptr = new double;
```

- new returns address of memory location

Dynamic Memory Allocation

如何要一個動態的陣列?

- Can also use `new` to allocate array

  ```
  arrayPtr = new double[25];
  ```

  – Program often terminates if there is no sufficient memory

- Can then use [ ] or pointer arithmetic to access array

Releasing Dynamic Memory

- Use `delete` to free dynamic memory

  ```
  delete dptr;
  ```

- Use `delete []` to free dynamic array memory

  ```
  delete [] arrayptr;
  ```

- Only use `delete` with dynamic memory!

Dangling Pointers and Memory Leaks

- A pointer is dangling if it contains the address of memory that has been freed by a call to `delete`.
  - Solution: set such pointers to 0 as soon as memory is freed.
- A memory leak occurs if no-longer-needed dynamic memory is not freed. The memory is unavailable for reuse within the program.
  - Solution: free up dynamic memory after use

10.10 Returning Pointers from Functions

- Pointer can be return type of function

```
int* newNum();
```

- Function must not return a pointer to a local variable in the function
- Function should only return a pointer
  - to data that was passed to the function as an argument
  - to dynamically allocated memory

10.11 Pointers to Class Objects and Structures

- Can create pointers to objects and structure variables

```
struct Student {…};
class Square {…};
Student stu1;
Student *stuPtr = &stu1;
Square sq1[4];
Square *squarePtr = &sq1[0];
```

- Need ( ) when using * and .

$\boxed{\texttt{(*stuPtr)}}$`.studentID = 12204;`

Object

Structure Pointer Operator

- Simpler notation than `(*ptr).member`
- Use the form `ptr->member`:

```
stuPtr->studentID = 12204;
squarePtr->setSide(14);
```

in place of the form `(*ptr).member`:

```
(*stuPtr).studentID = 12204;
(*squarePtr).setSide(14);
```

Dynamic Memory with Objects

- Can allocate dynamic structure variables and objects using pointers:

```
stuPtr = new Student;
```

- Can pass values to constructor:

```
squarePtr = new Square(17);
```

- `delete` causes destructor to be invoked:

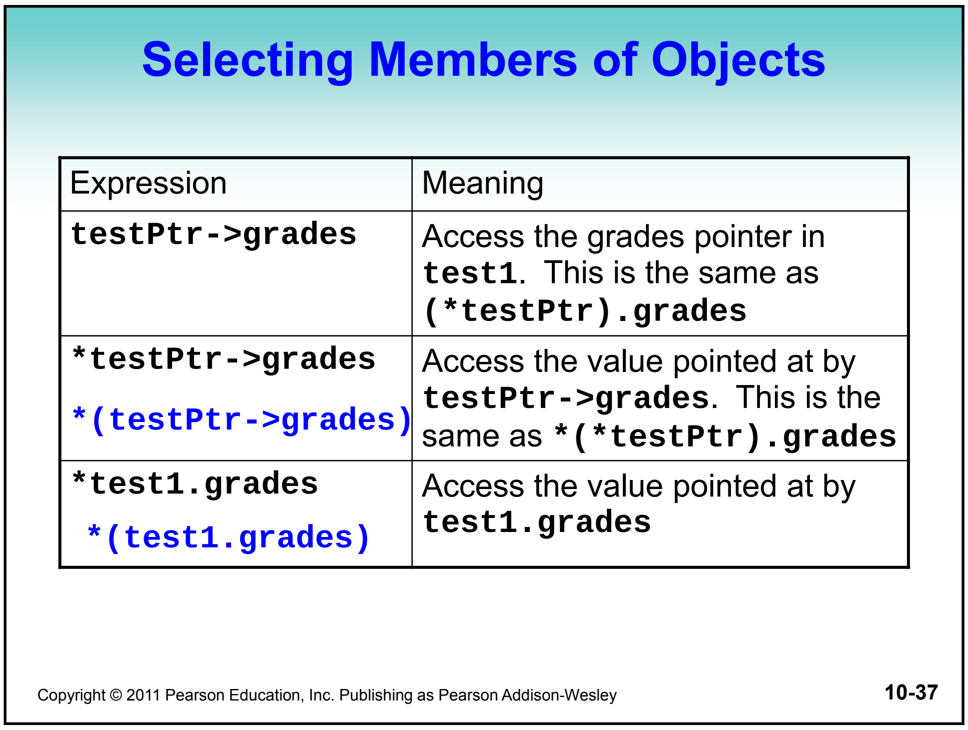
```
delete squarePtr;
```

10.12 Selecting Members of Objects

Situation: A structure/object contains a pointer as a member. There is also a pointer to the structure/ object.

Problem: How do we access the pointer member via the structure/object pointer?

```
struct GradeList
  { string courseNum;
    int * grades;
  }
GradeList test1, *testPtr = &test1;
```



Selecting Members of Objects

Expression	Meaning
`testPtr->grades`	Access the grades pointer in `test1`. This is the same as `(*testPtr).grades`

`*testPtr->grades`
`*(testPtr->grades)`
	Access the value pointed at by `testPtr->grades`. This is the same as `*(*testPtr).grades`

`*test1.grades`
`*(test1.grades)`
	Access the value pointed at by `test1.grades`

Chapter 11: More About Classes and Object-Oriented Programming

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

Topics

11.1 The `this` Pointer and Constant Member Functions
11.2 Static Members
11.3 Friends of Classes
11.4 Memberwise Assignment
11.5 Copy Constructors
11.6 Operator Overloading
11.7 Type Conversion Operators

Topics (continued)

11.8 Convert Constructors
11.9 Aggregation and Composition
11.10 Inheritance
11.11 Protected Members and Class Access
11.12 Constructors, Destructors, and Inheritance
11.13 Overriding Base Class Functions

11.1 The `this` Pointer and Constant Member Functions

- `this` pointer: 本地 object 的 pointer
  - Implicit parameter passed to a member function
  - points to the object calling the function
- `Const` member function:
  - does not modify its calling object

Using the this Pointer

Can be used to access members that may be hidden by parameters with the same name:

```
class SomeClass
{
  private:
    int num;
  public:
    void setNum(int num)
    { this->num = num; }
};
```

引數 “*num*” 與 data member “*num*” 同名
➔ data member “*num*”被遮蔽

Constant Member Functions

- Declared with keyword `const`
- When const follows the parameter list,

  `int getX()const` getX()不可更改所屬的 object

  the function is prevented from modifying the object.
- When const appears in the parameter list,

  `int setNum (const int num)`

  the function is prevented from modifying the parameter. The parameter is read-only.

setNum()不可更改引數 num

11.2 Static Members

Ex: 網頁的記數器

- Static member variable:
  - One instance of variable for the entire class
  - Shared by all objects of the class
- Static member function: 相關於 Class ，而非 Object
  - Can be used to access static member variables
  - Can be called before any class objects are created

Static Member Variables

(1) Must be declared in class with keyword `static`:

```
class IntVal
{
  public:
     static int valCount;
     intVal(int val = 0)
     { value = val; valCount++; }
     int getVal();
     void setVal(int);
   private:
     int value;
};
```

Shared by all *Objects* of Class *IntVal*

Static Member Variables

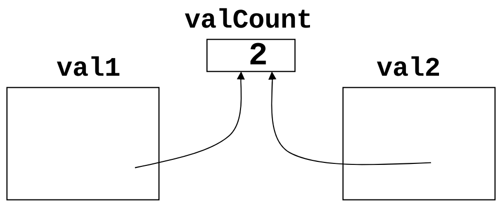
(2) Must be re-defined outside of the class:

```
class IntVal
{
    //In-class declaration
    static int valCount;
    //Other members not shown
 };
 //Re-Definition outside of class
 int IntVal::valCount = 0;
```

Static Member Variables

3) Can be accessed or modified by any object of the class: Modifications by one object are visible to all objects of the class:

```
IntVal val1, val2;
```



Static Member Functions

1) Declared with `static` as the return type:

```
class IntVal
{ public:
    static int getValCount()
    { return valCount; }
  private:
    int value;
    static int valCount;
};
```

`getValCount()` is a static member function

Static Member Functions

2) Can be called independently of class objects, through the class name:

```
cout << IntVal::getValCount();
```

3) Because of item 2 above, the `this` pointer cannot be used

4) Can be called before any objects of the class have been created

5) Used mostly to manipulate static member variables of the class

11.3 Friends of Classes

- **Friend function**: a function that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with the `friend` keyword in the function prototype

Friend Function Declarations

1) Friend function may be a stand-alone function:

```
class aClass
{
  private:
    int x;
    friend void fSet(aClass &c, int a);
};
```

```
void fSet(aClass &c, int a)
{
   c.x = a;
}
```

Stand-alone
Friend Function

Friend Function Declarations

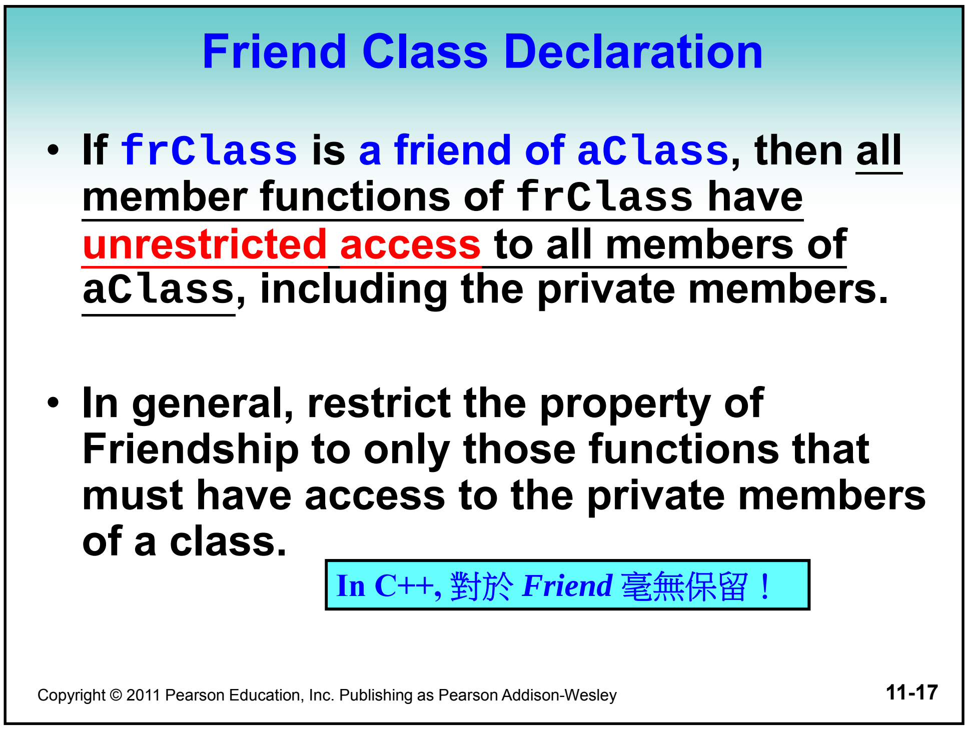
2) Friend function may be a member of another class:

```
class aClass
{ private:
   int x;
   friend void OtherClass::fSet
                    (aClass &c, int a);
};
class OtherClass
{ public:
    void fSet(aClass &c, int a)
    { c.x = a; }
};
```

Friend Class Declaration

3) An entire class can be declared a friend of a class:

```
class aClass
{private:
   int x;
   friend class frClass;
};
class frClass
{
 public:
  void fSet(aClass &c,int a){c.x = a;}
  int fGet(aClass c){return c.x;}
};
```

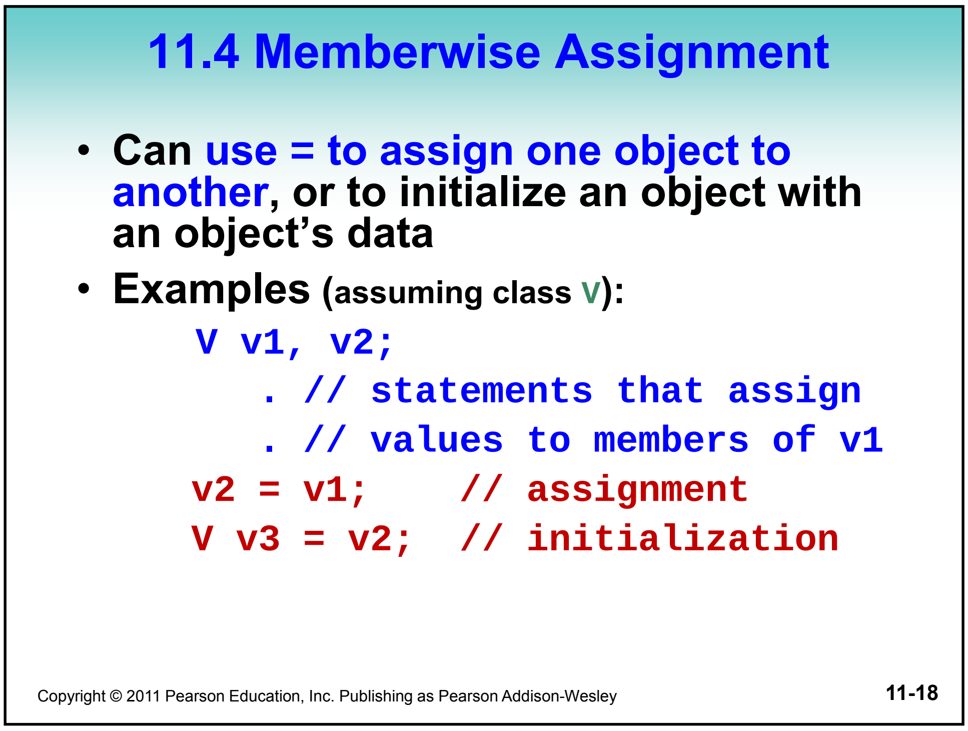


Friend Class Declaration

- If `frClass` is a friend of `aClass`, then all member functions of `frClass` have unrestricted access to all members of `aClass`, including the private members.

- In general, restrict the property of Friendship to only those functions that must have access to the private members of a class.

In C++, 對於 *Friend* 毫無保留！



11.4 Memberwise Assignment

- Can use = to assign one object to another, or to initialize an object with an object's data
- Examples (assuming class `V`):

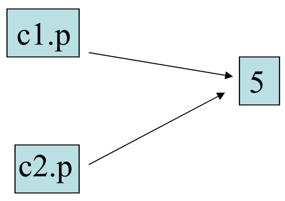
```
V v1, v2;
   . // statements that assign
   . // values to members of v1
v2 = v1;    // assignment
V v3 = v2;  // initialization
```

11.5 Copy Constructors

- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor copies field-to-field
- Default copy constructor works fine in many cases

Default Constructor Causes Sharing of Storage

```
CpClass c1(5);
if (true)
{
   CpClass c2;
   c2 = c1; 較不安全的 copy
}
// c1 is corrupted
// when c2 goes
// out of scope when
// its destructor
// executes
```



Problems of Sharing Dynamic Storage - Dangerous!

- Destructor of one object deletes memory still in use by other objects
- Modification of memory by one object affects other objects sharing that memory

Copy Constructors (when dynamic memory allocation)

Problems occur when objects contain pointers to dynamic storage:

(FIX: use a self-defined *constructor*)

```
class CpClass
{
 private:
  int *p;

 public:
  CpClass(int v=0) Constructor function
    { p = new int; *p = v;}
  ~CpClass(){delete p;} Destructor function
};
```

Programmer-Defined Copy Constructors

- A copy constructor can be one that takes a "reference parameter" to another object of the same class
- The copy constructor uses the data in the object passed as parameter to initialize the object being created
- Reference parameter should be `const` to avoid potential for data corruption

Programmer-Defined Copy Constructors

- The copy constructor avoids problems caused by memory sharing
- Can allocate separate memory to hold new object's dynamic member data
- Can make new object's pointer point to this memory
- Copies the data, not the pointer, from the original object to the new object

Copy Constructor Example

Copy Constructor

```
class CpClass
{
  int *p;

  public:
    CpClass(const CpClass &obj)
    { p = new int; *p = *obj.p; }

    CpClass(int v=0)
    { p = new int; *p = v; }

    ~CpClass(){delete p;}
};
```

11.6 Operator Overloading

- Operators such as =, +, and others can be redefined for use with objects of a class
- The name of the function for the overloaded operator is `operator` followed by the operator symbol, *e.g.*,
  - `operator+` is the overloaded + operator and
  - `operator=` is the overloaded = operator

Operator Overloading

- Operators can be overloaded as
  - instance member functions or as
  - friend functions
- Overloaded operator must have the same number of parameters as the standard version. For example, `operator=` must have two parameters, since the standard = operator takes two parameters.

Overloading Operators

A binary operator overloaded as an instance member needs only one parameter, which represents the operand on the right:

```
class OpClass
{
 private:
   int x;
 public:
   OpClass operator+(OpClass right);
};
```

The local object is used as the left operand

Overloading Operators

- The left operand of the overloaded binary operator is the calling object
- The implicit left parameter is accessed through the `this` pointer

```
OpClass OpClass::operator+(OpClass r)
{  OpClass sum;
   sum.x = this->x + r.x;
   return sum;
}
```

(Calling Object) + (Input Object)

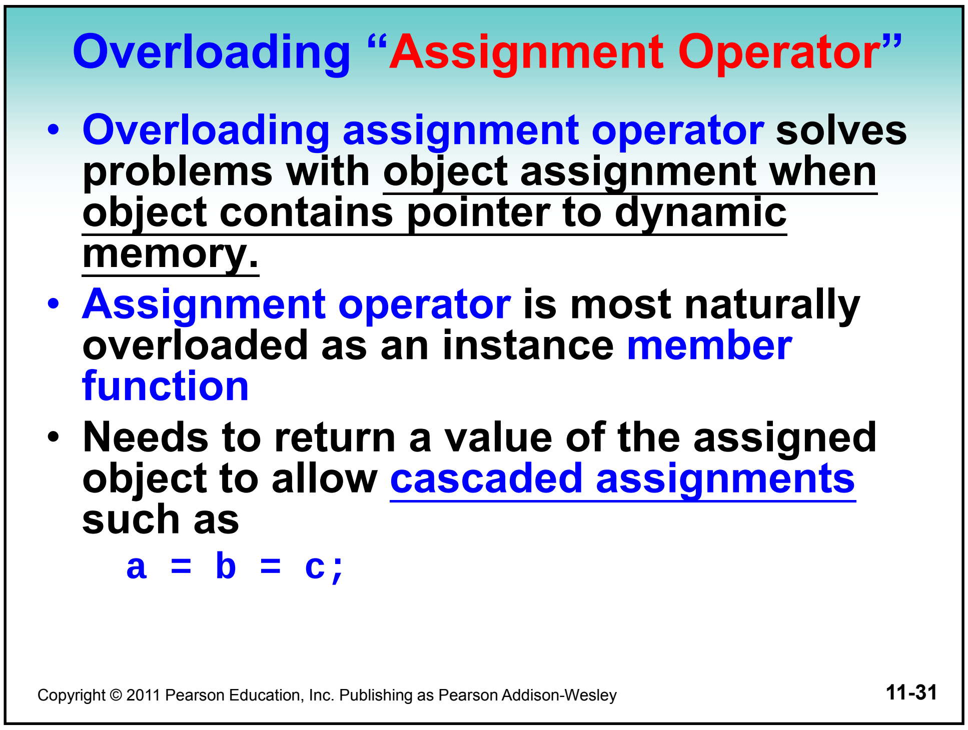
Invoking an Overloaded Operator

- Operator can be invoked as a member function:

```
OpClass a, b, s;
s = a.operator+(b);
```

- It can also be invoked in the more conventional manner:

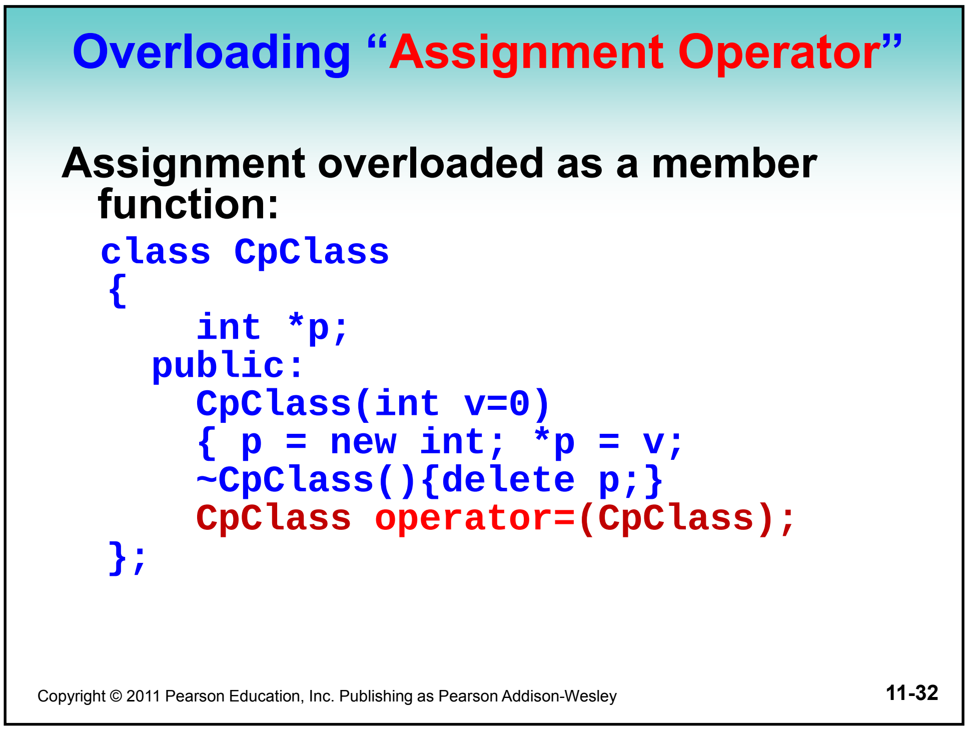
```
OpClass a, b, s;
s = a + b;
```



Overloading “Assignment Operator”

- Overloading assignment operator solves problems with object assignment when object contains pointer to dynamic memory.
- Assignment operator is most naturally overloaded as an instance member function
- Needs to return a value of the assigned object to allow cascaded assignments such as

```
a = b = c;
```



Overloading “Assignment Operator”

Assignment overloaded as a member function:

```
class CpClass
{
    int *p;
  public:
    CpClass(int v=0)
    { p = new int; *p = v;
    ~CpClass(){delete p;}
    CpClass operator=(CpClass);
};
```

Overloading “Assignment Operator”

Implementation returns a value:

```
CpClass CpClass::operator=(CpClass r)
{
   *p = *r.p;
   return *this;
};
```

Invoking the assignment operator:

```
CpClass a, x(45);
a.operator=(x); // either of these
a = x;          // lines can be used
```

Notes on Overloaded Operators

- Can change the entire meaning of an operator
- Most operators can be overloaded
- Cannot change the number of operands of the operator
- Cannot overload the following operators:

```
?:   .   .*   sizeof
```

Overloading Types of Operators

- ++, -- operators overloaded differently for prefix vs. postfix notation

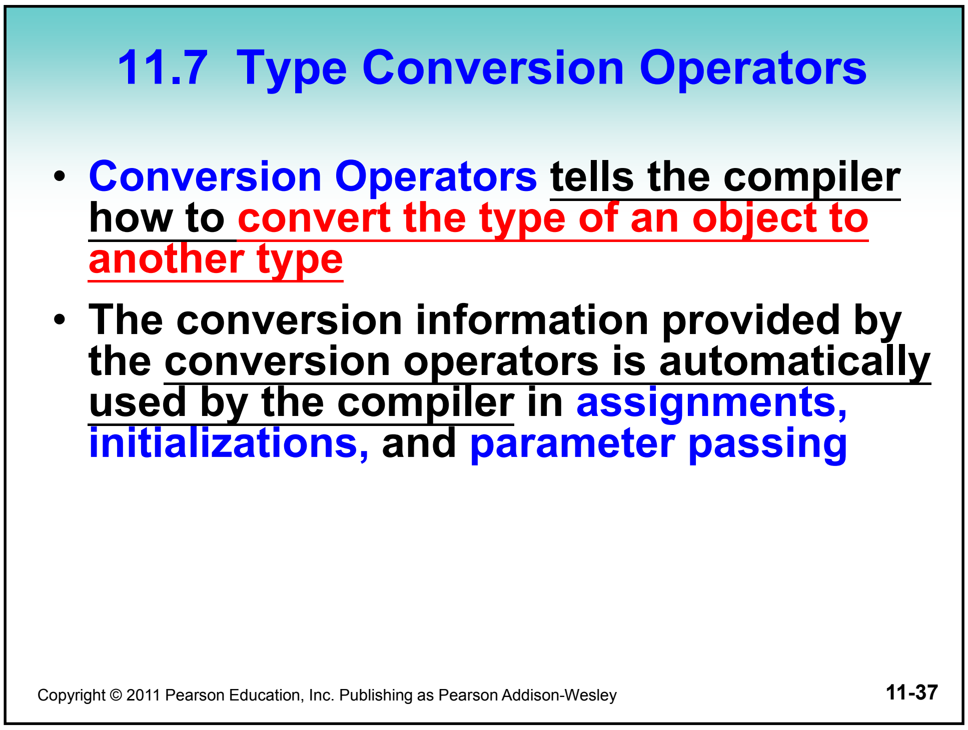
```
Prefix++ overloading: operator++()
Postfix++ overloading: operator++(int) // int is not integer here
```

- Overloaded relational operators should return a `bool` value
- Overloaded stream operators >>, << must return `istream`, `ostream` objects and take `istream`, `ostream` objects as parameters

Overloaded [] Operator

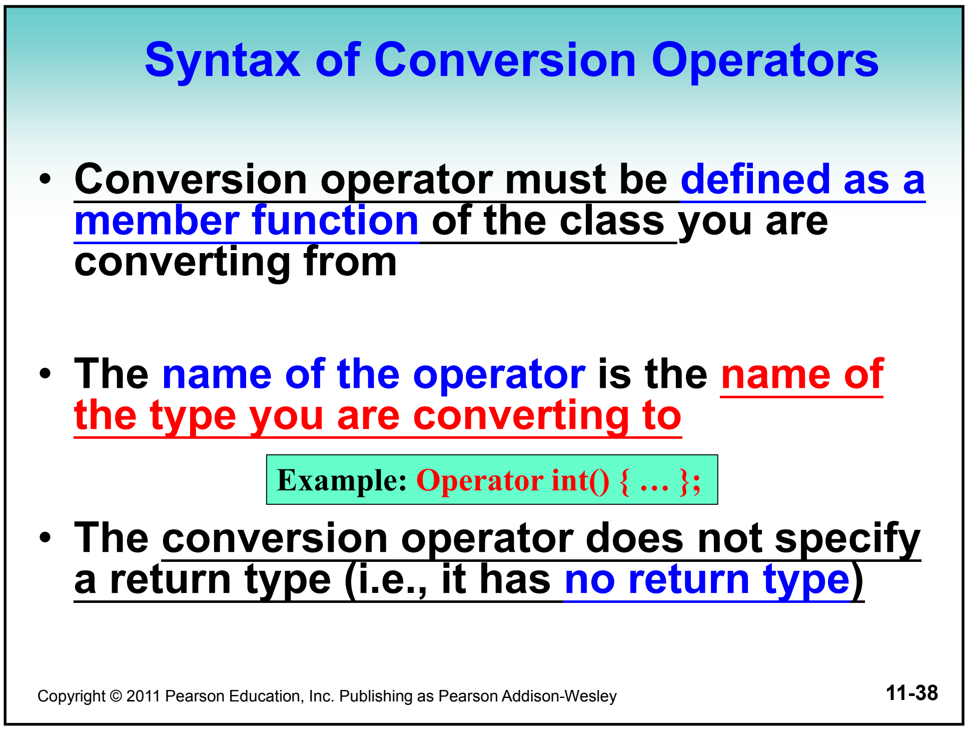
- Can be used to create classes that behave like arrays, providing bound-checking on subscripts

- Overloaded [] returns a reference to object, not an object itself



11.7 Type Conversion Operators

- Conversion Operators tells the compiler how to convert the type of an object to another type
- The conversion information provided by the conversion operators is automatically used by the compiler in assignments, initializations, and parameter passing



Syntax of Conversion Operators

- Conversion operator must be defined as a member function of the class you are converting from
- The name of the operator is the name of the type you are converting to

Example: `Operator int() { … };`

- The conversion operator does not specify a return type (i.e., it has no return type)

Conversion Operator Example

- To convert from a class `IntVal` to an integer:

```
class IntVal
{
 int x;
 public:
   IntVal(int a = 0){x = a;}
   operator int(){return x;}
};
```

- Automatic conversion during assignment:

```
IntVal obj(15); int i;
i = obj;  cout << i; // prints 15
```

自動轉態

11.8 Convert Constructors

Convert constructors are constructors with a single parameter of a type other than the class

```
class CCClass
{
 int x;
  public:
    CCClass()           //default
    CCClass(int a, int b);
    CCClass(int a);     //convert
    CCClass(string s);  //convert
};
```

Example of a Convert Constructor

The C++ `string` class has a convert constructor that converts from C-strings:

```
class string
{
  public:
    string(char *);  //convert
    …
};
```

輸入引數是 C-string
轉為 C++ string

Uses of Convert Constructors

- Automatically invoked by the compiler to create an object from the value passed as parameter:

```
string s("hello");  //convert C-string
CCClass obj(24);    //convert int
```

- Compiler allows convert constructor to be invoked with assignment-like notation:

```
string s = "hello"; //convert C-string
CCClass obj = 24;   //convert int
```

Uses of Convert Constructors

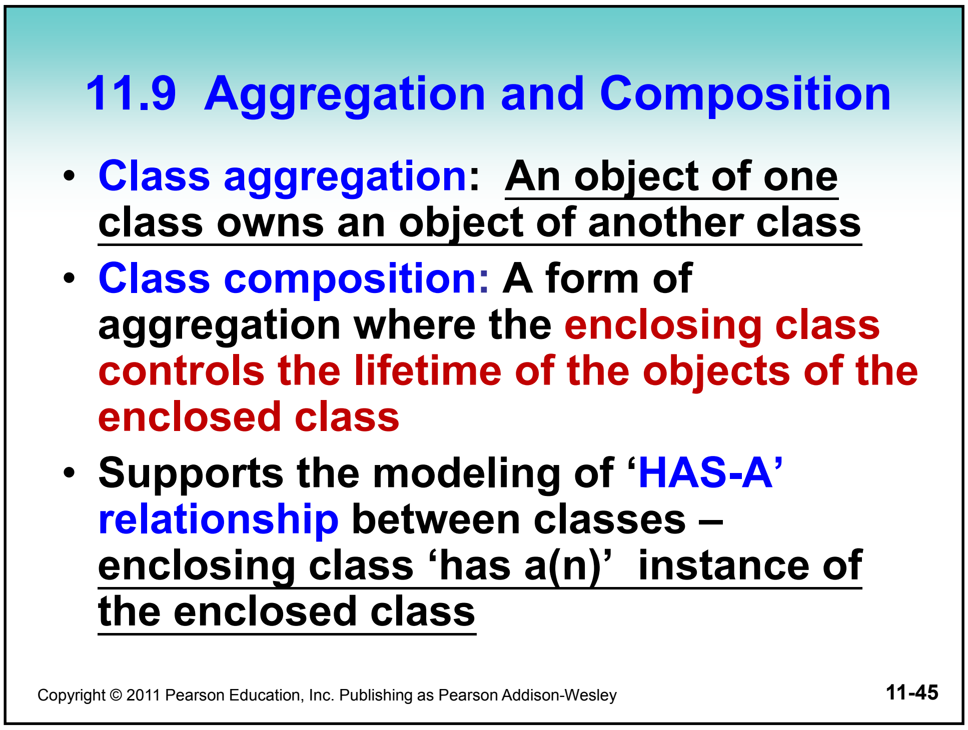
- Convert constructors allow functions to take a parameter of <u>not specified type</u>:

```
void myFun(string s); // needs string
                      // object
myFun("hello");       // accepts C-string

void myFun(CCClass c);
myFun(34);            // accepts int
```

自動先把 34 轉為 CCClass Object，再執行 myFun()

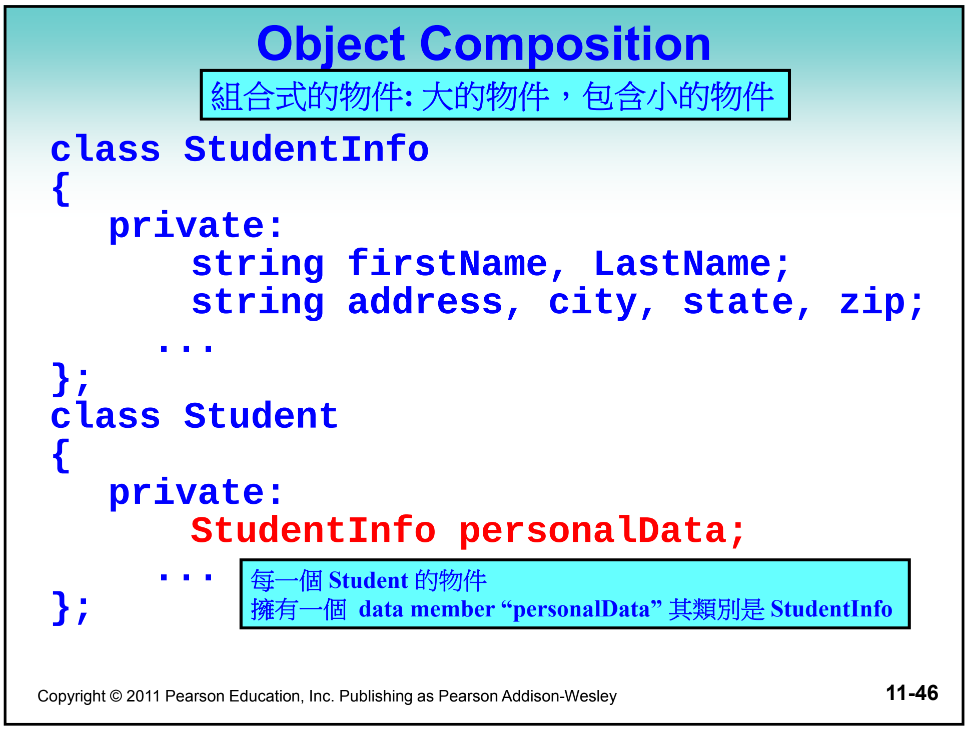
Topics (continued)

11.8 Convert Constructors
➡ 11.9 Aggregation and Composition
11.10 Inheritance
11.11 Protected Members and Class Access
11.12 Constructors, Destructors, and Inheritance
11.13 Overriding Base Class Functions



11.9 Aggregation and Composition

- Class aggregation: An object of one class owns an object of another class
- Class composition: A form of aggregation where the enclosing class controls the lifetime of the objects of the enclosed class
- Supports the modeling of 'HAS-A' relationship between classes – enclosing class 'has a(n)' instance of the enclosed class



Object Composition

組合式的物件: 大的物件，包含小的物件

```
class StudentInfo
{
   private:
       string firstName, LastName;
       string address, city, state, zip;
     ...
};
class Student
{
   private:
       StudentInfo personalData;
     ...
};
```

每一個 Student 的物件
擁有一個 data member "personalData" 其類別是 StudentInfo

Member Initialization Lists

大的物件的起始，包含小的物件的 constructor 所需之參數

- Used in constructors for classes involved in aggregation.
- Allows constructor for enclosing class to pass arguments to the constructor of the enclosed class
- Notation:

```
owner_class(parameters):owned_class(parameters);
```

Member Initialization Lists

Use:

```
class StudentInfo
{
     ...
};
class Student
{
   private:
      StudentInfo personalData;
   public:
      Student(string fname, string lname):
      personalData(fname, lname);
};
```

階層式或組合式的資料結構，
StudentInfo 是小物件的Class，
Student 是較大的物件Class。

Initialization List 的一行指令

Member Initialization Lists

- Member Initialization lists can be used to simplify the coding of constructors
- Should keep the <u>entries in the initialization list</u> in the <u>same order</u> as they are declared in the class

Aggregation Through Pointers

- A ‘HAS-A’ relationship can be implemented by <u>owning a pointer to an object</u>
- Can be used when multiple objects of a class may ‘have’ the same attribute for a member
  - ex: students who may have the same city/state/zipcode
- Using pointers minimizes data duplication and saves space

Aggregation, Composition, and Object Lifetimes

上層(大的) 下層(小的)

- Aggregation represents the owner/owned relationship between objects.
- Composition is a form of aggregation in which the lifetime of the owned object is the same as that of the owner object
- Owned object is usually created as part of the owning object's constructor, destroyed as part of owning object's destructor

11.10 Inheritance

一個衍生性 (Derived) 的物件，繼承原生性 (Base) 的物件一些 members

- Inheritance is a way of creating a new class by starting with an existing class and adding new members
- The new class can replace or extend the functionality of the existing class
- Inheritance models the 'IS-A' relationship between classes

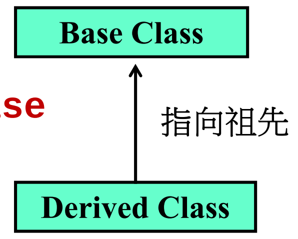
Inheritance – Terminology

- The existing class is called the **base class**
  – Alternates: **parent class, superclass**
- The new class is called the **derived class**
  – Alternates: **child class, subclass**

Inheritance Syntax and Notation

```
// Existing class
class Base
{
};
// Derived class
class Derived : public Base
{
};
```

Inheritance Class Diagram



Inheritance of Members

```
class Parent
{
  int a;
  void bf();
};
class Child : public
        Parent
{
  int c;
  void df();
};
```

Objects of "Parent" have members

```
int a; void bf();
```

Objects of "Child" have members

```
int a; void bf();
int c; void df();
```

a, bf() in *Child*, 不用宣告就有了

11.11 Protected Members and Class Access

- protected member access specification: A class member labeled `protected` is accessible by member functions of derived classes as well as by member functions of the same class
- Like `private`, but also accessible by member functions of derived classes

“Base Class” Access Specification

Base class access specification: determines how `private`, `protected`, and `public` members of base class can be accessed by derived classes

“Base Class” Access

C++ supports three inheritance modes, also called base class access modes:

- public inheritance
```
class Child : public Parent { };
```
- protected inheritance
```
class Child : protected Parent{ };
```
- private inheritance
```
class Child : private Parent{ };
```

"Base Class" Access vs. Member Access Specification

Base class access not the same as member access specification:

- Base class access: determine access for inherited members
- Member access specification: determine access for members defined in the class

Member Access Specification

Specified using the keywords `private`, `protected`, `public`

```
class MyClass
{
  private: int a;
  protected: int b; void fun();
  public: void fun2();
};
```

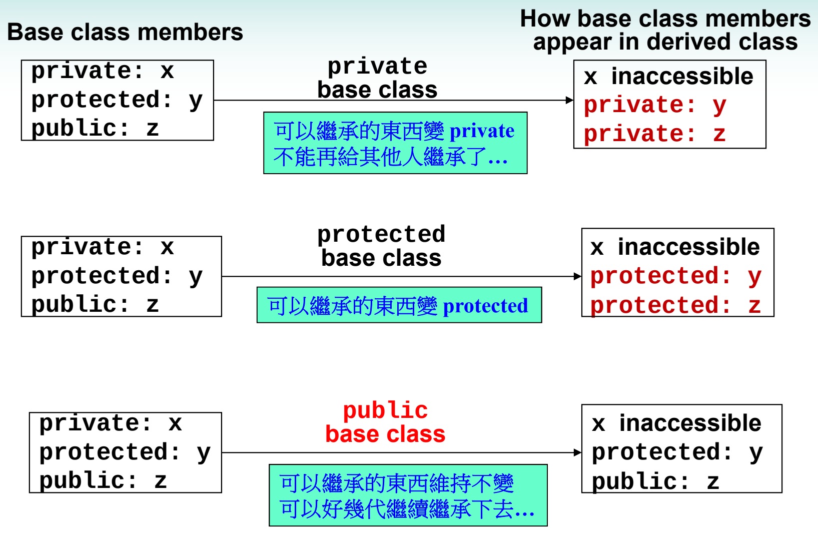
Base Class Access Specification

```
class Child : public Parent
{
   protected:             base access
      int a;
   public:
      Child();            member access
};
```

"Base Class" Access Specifiers

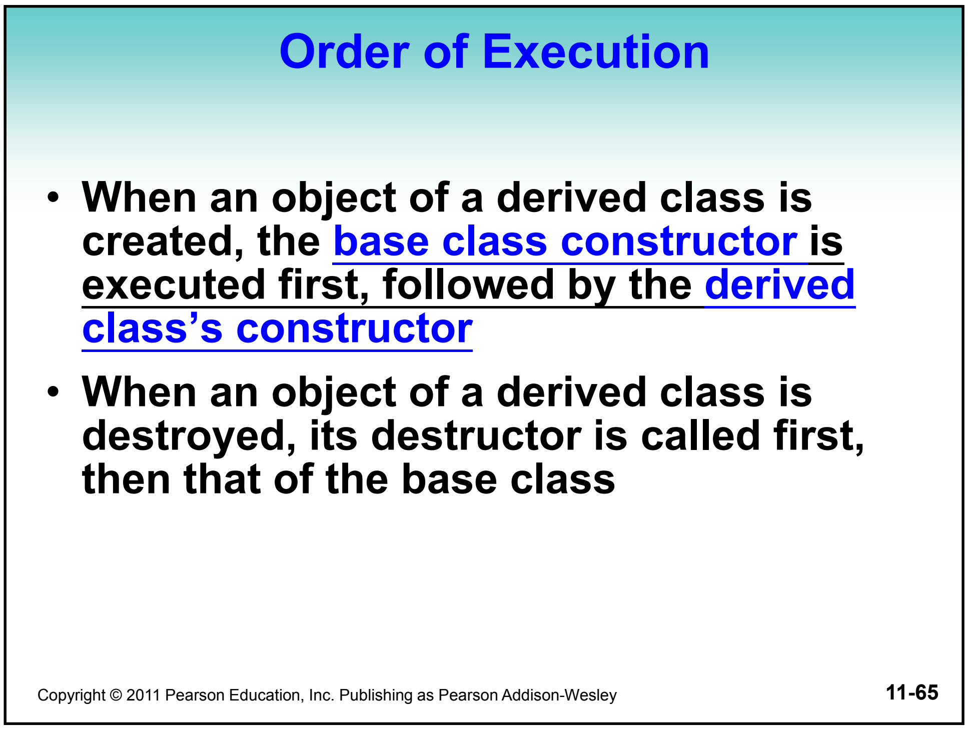
1) `public` – object of derived class can be treated as object of base class (not vice-versa)
2) `protected` – more restrictive than `public`, but allows derived classes to know some of the details of parents
3) `private` – prevents objects of derived class from being treated as objects of base class.

Effect of Base Access



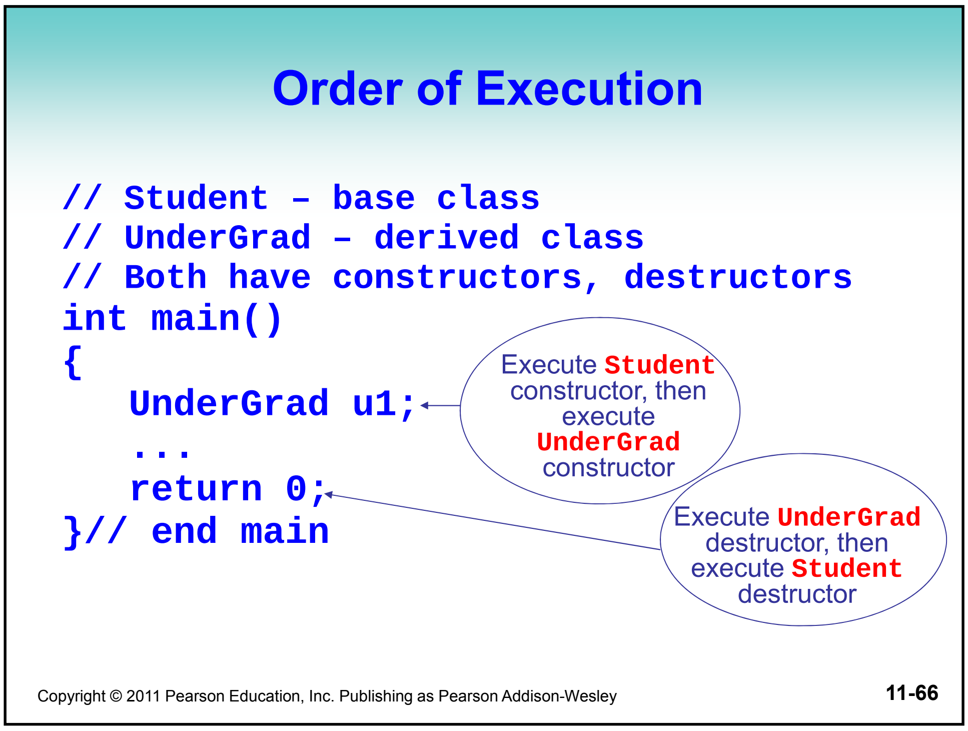
11.12 Constructors, Destructors and Inheritance

- By inheriting every member of the base class, a derived class object contains a base class object
- The derived class constructor can specify which base class constructor should be used to initialize the base class object



Order of Execution

- When an object of a derived class is created, the base class constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class



Order of Execution

```
// Student – base class
// UnderGrad – derived class
// Both have constructors, destructors
int main()
{
   UnderGrad u1;
   ...
   return 0;
}// end main
```

Execute Student constructor, then execute UnderGrad constructor

Execute UnderGrad destructor, then execute Student destructor

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading
- Must be done if base class has no default constructor

Passing Arguments to Base Class Constructor

```
class Parent {
    int x, y;
    public: Parent(int,int);
};
class Child : public Parent {
    int z
    public:
    Child(int a): Parent(a,a*a)
    {z = a;}
};
```

11.13 Overriding Base Class Functions

Overridding functions 就是 derived class 中被重新定義的 functions

- Overriding: function in a derived class that has the *same name and parameter list* as a function in the base class
- Typically used to replace a function in base class with different actions in derived class
- Not the same as overloading – with overloading, the parameter lists must be different

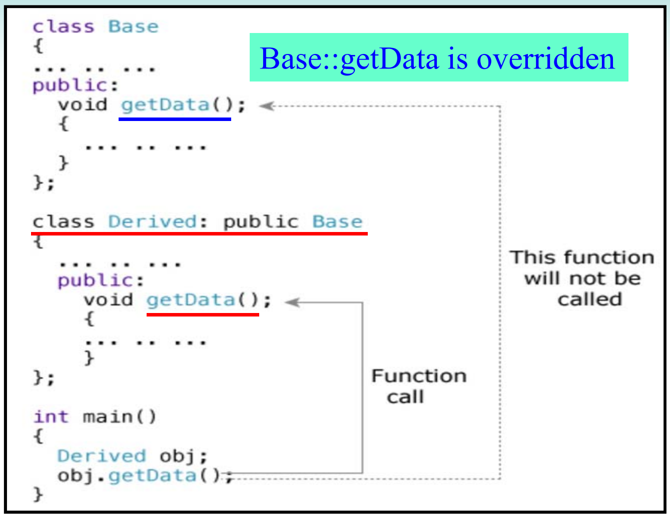
Access to Overridden Function

- When a function is overridden, all objects of derived class use the overriding function (i.e., the new function).
- If necessary to access the overridden version of the function, it can be done using the scope resolution operator with the name of the base class and the name of the function:

```
Student::getName();
```

呼叫舊的 function in the base class, *Student*

Overridden Function



```
class Base
{
... .. ...
public:
  void getData();
  {
    ... .. ...
  }
};

class Derived: public Base
{
  ... .. ...
  public:
    void getData();
    {
    ... .. ...
    }
};

int main()
{
  Derived obj;
  obj.getData();
}
```

Chapter 12: More About Characters, Strings, and the `string` Class

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

Topics

12.1 C-Strings
12.2 Library Functions for Working with C-Strings
12.3 Conversions Between Numbers and Strings
12.4 Character Testing

Topics (continued)

12.5 Character Case Conversion

12.6 Writing Your Own C-String Handling Functions

12.7 More About the C++ string Class

12.8 Creating Your Own String Class

12.1 C-Strings

- C-string: sequence of characters stored in adjacent memory locations and terminated by `NULL` character
- The C-string

  `"Hi there!"`

  would be stored in memory as shown:

`H`	`i`		`t`	`h`	`e`	`r`	`e`	`!`	`\0`

Representation of C-strings

As a string literal

```
"Hi There!"
```

As a pointer to char

```
char *p;
```

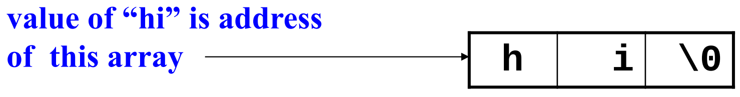
As an array of characters

```
char str[20];
```

All three representations are pointers to char

String Literals

- A string literal is stored as a null-terminated array of char
- Compiler uses the address of the first character of the array as the value of the string
- String literal is a pointer to char



Array of char

- Array of char can be defined and initialized to a C-string

```
char str1[20] = "hi";
```

- Array of char can be defined and later have a string copied into it

```
char str2[20];
strcpy(str2, "hi");
```

目的地字串，str2, 需自備空間

Array of char

- Name of array of char is used as a pointer to char
- Unlike string literal, a C-string defined as an array can be referred to in other parts of the program by using the array name

Pointer to char

- Defined as

```
    char *pStr;
```

字元指標而已，無空間

- Does not itself allocate memory
- Useful in repeatedly referring to C-strings defined as a string literal

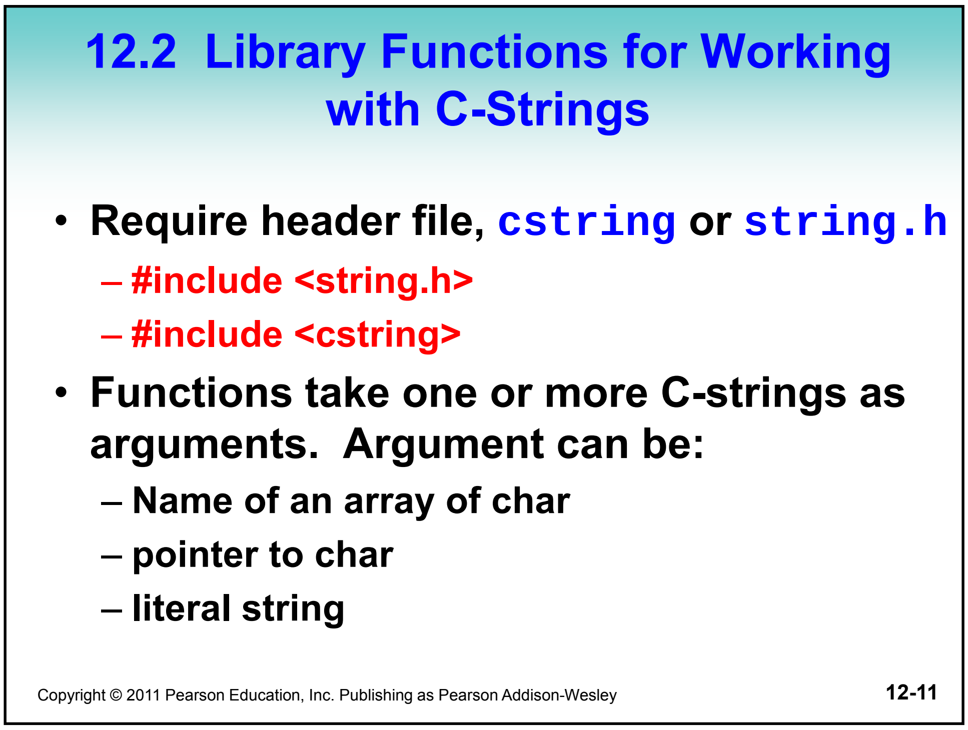
```
    pStr = "Hi there";
    cout << pStr << " "
         << pStr;
```

Pointer to char

- Pointer to char can also refer to C-strings defined as arrays of char

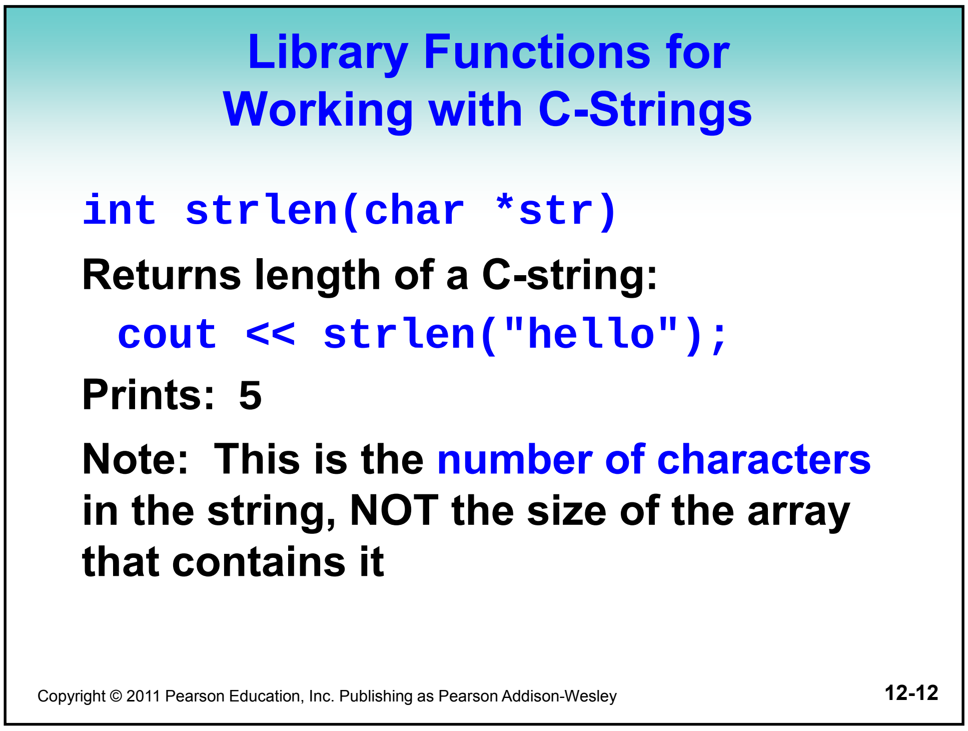
```
    char str[20] = "hi";
    char *pStr = str;
    cout << pStr;  // prints hi
```

- Can dynamically allocate memory to be used for C-string using new



12.2 Library Functions for Working with C-Strings

- Require header file, `cstring` or `string.h`
  - #include <string.h>
  - #include <cstring>
- Functions take one or more C-strings as arguments. Argument can be:
  - Name of an array of char
  - pointer to char
  - literal string



Library Functions for Working with C-Strings

```
int strlen(char *str)
```

Returns length of a C-string:

```
  cout << strlen("hello");
```

Prints: 5

Note: This is the number of characters in the string, NOT the size of the array that contains it

Strcat

兩個字串串接

`strcat(char *dest, char *source)`

- Takes two C-strings as input. It adds the contents of the second string to the end of the first string:

```
char str1[15] = "Good ";
char str2[30] = "Morning!";
strcat(str1, str2);
cout << str1; // prints: Good Morning!
```

- No automatic bounds checking: programmer must ensure that str1 has enough room for the result

Strcpy

字串複製

`strcpy(char *dest, char *source)`

Copies a string from a source address to a destination address

```
char name[15];
strcpy(name, "Deborah");
cout << name; // prints Deborah
```

Strcmp

字串比較

`int strcmp(char *str1, char*str2)`

- Compares strings stored at two addresses to determine their relative alphabetic order:
- Returns a value:
  - less than 0 if `str1` precedes `str2`
  - equal to 0 if `str1` equals `str2`
  - greater than 0 if `str1` succeeds `str2`

Strcmp

- Often used to test for equality

```
  if(strcmp(str1, str2) == 0)
     cout << "equal";
 else
     cout << "not equal";
```

- Also used to determine ordering of C-strings in sorting applications
- Note that C-strings cannot be compared using ==, (which compares addresses of C-strings, not contents)

Strstr

子字串搜尋

`char *strstr(char *str1,char *str2)`

- Searches for the occurrence of `str2` within `str1`.
- Returns a pointer to the occurrence of `str2` within `str1` if found, and returns `NULL` otherwise

```
char s[15] = "Abracadabra";
char *found = strstr(s,"dab");
cout << found;     // prints dabra
```

12.3 Conversions Between Numbers and Strings

- These classes that can be used to convert between string and numeric forms of numbers
- Need to include `sstream` header file

ss.str("");
ss.clear();

Conversion Classes 字串河流

- `istringstream`:
  - contains a string to be converted to numeric values where necessary
  - Use `str(s)` to initialize string to contents of s
  - Use the stream extraction operator >> to read from the string 輸入
- `ostringstream`:
  - collects a string in which numeric data is converted as necessary
  - Use the stream insertion operator << to add data onto the string 輸出
  - Use `str()` to retrieve converted string

Example:
stringstream ss; float num;
ss.clear(); ss.str(“123.999”);
ss >> num; // num will be 123.999

atoi and atol

字串轉整數

- `atoi` converts alphanumeric to int
- `atol` converts alphanumeric to long

```
int atoi(char *numericStr)
 long atol(char *numericStr)
```

- Examples:

```
int number; long lnumber;
number = atoi("57");
lnumber = atol("50000");
```

Atof

字串轉浮點數

- atof converts a numeric string to a floating point number, actually a double

```
double atof(char *numericStr)
```

- Example:

```
double dnumber;
dnumber = atof("3.14159");
```

atoi, atol, atof

字串有非法的 **non-digit** 字元時…

- if C-string being converted contains non-digit characters, results are undefined
  - function may return result of conversion up to first non-digit
  - function may return 0

- All functions require cstdlib

itoa

整數轉成字串，可指定基數 (Digit Base)

- itoa converts an int to an alphanumeric string
- Allows user to specify the base of conversion

```
itoa(int num, char *numStr, int base)
```

- Example: To convert the number 1200 to a hexadecimal string

```
char numStr[10];
itoa(1200, numStr, 16);
```

- The function performs no bounds-checking on the array numStr

12.4 Character Testing

require cctype header file

FUNCTION	MEANING
isalpha	**true** if arg. is a letter, **false** otherwise
isalnum	**true** if arg. is a letter or digit, **false** otherwise
isdigit	**true** if arg. is a digit 0-9, **false** otherwise
islower	**true** if arg. is lowercase letter, **false** otherwise

Character Testing

require `cctype` header file

FUNCTION	MEANING
`isprint`	`true` if arg. is a **printable character**, `false` otherwise
`ispunct`	`true` if arg. is a **punctuation** character, `false` otherwise
`isupper`	`true` if arg. is an uppercase letter, `false` otherwise
`isspace`	`true` if arg. is a whitespace character, `false` otherwise

12.5 Character Case Conversion

- **require `cctype` header file**
- **Functions:**
  - `toupper`: **convert a letter to uppercase equivalent**
  - `tolower`: **convert a letter to lowercase equivalent**

toupper

`toupper`: if char argument is lowercase letter, return uppercase equivalent; otherwise, return input unchanged

`toupper` actually takes an integer parameter and returns an integer result. The integers are the ascii codes of the characters

toupper

The function

```
   char upCase(int i)
   {return toupper(i);}
```

will work as follows:

```
char s[] = "Hello!";
cout << upCase(s[0]); //displays 'H'
cout << upCase(s[1]); //displays 'E'
cout << upCase(s[2]); //displays ‘L'
cout << upCase(s[3]); //displays ‘L'
cout << upCase(s[4]); //displays ‘O'
cout << upCase(s[5]); //displays '!'
```

tolower

`tolower`: if char argument is uppercase letter, return lowercase equivalent; otherwise, return input unchanged

Tolower

The function

```
   char loCase(int i)
   {return tolower(i);}
```

will work as follows

```
char s[] = "Hello!";
cout << loCase(s[0]); //displays 'h'
cout << loCase(s[1]); //displays 'e'
cout << loCase(s[2]); //displays ‘l'
cout << loCase(s[3]); //displays ‘l'
cout << loCase(s[4]); //displays ‘o'
cout << loCase(s[5]); //displays '!'
```

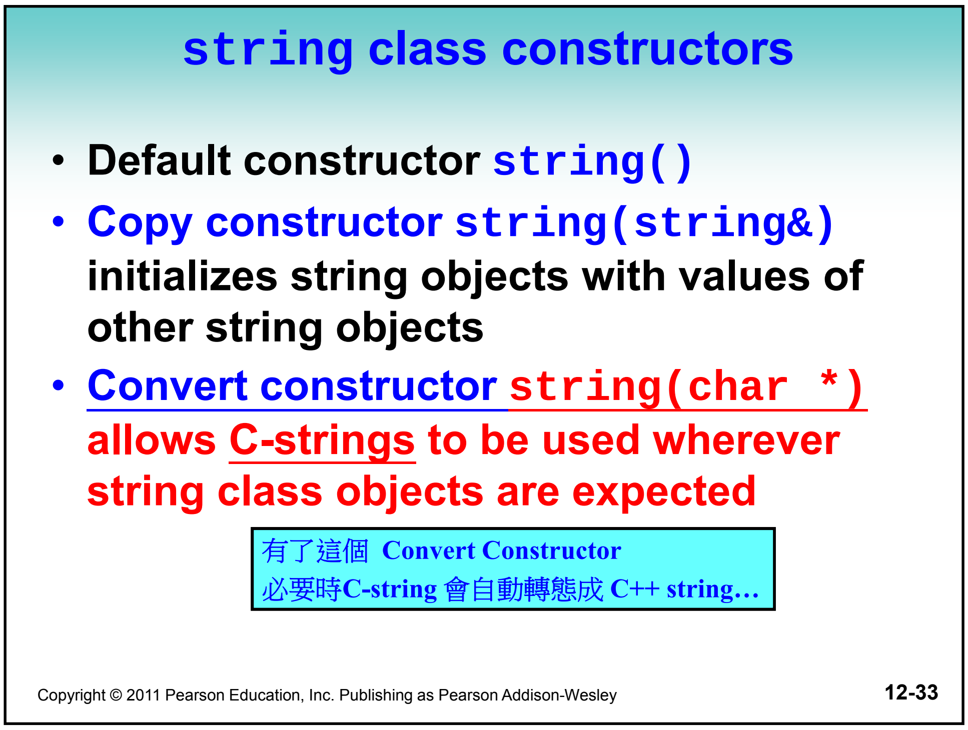
12.6 Writing Your Own C-String Handling Functions

When writing C-String Handling Functions:

- can pass arrays or pointers to char
- can perform bounds checking to ensure enough space for results
- can anticipate unexpected user input

12.7 More About the C++ `string` Class

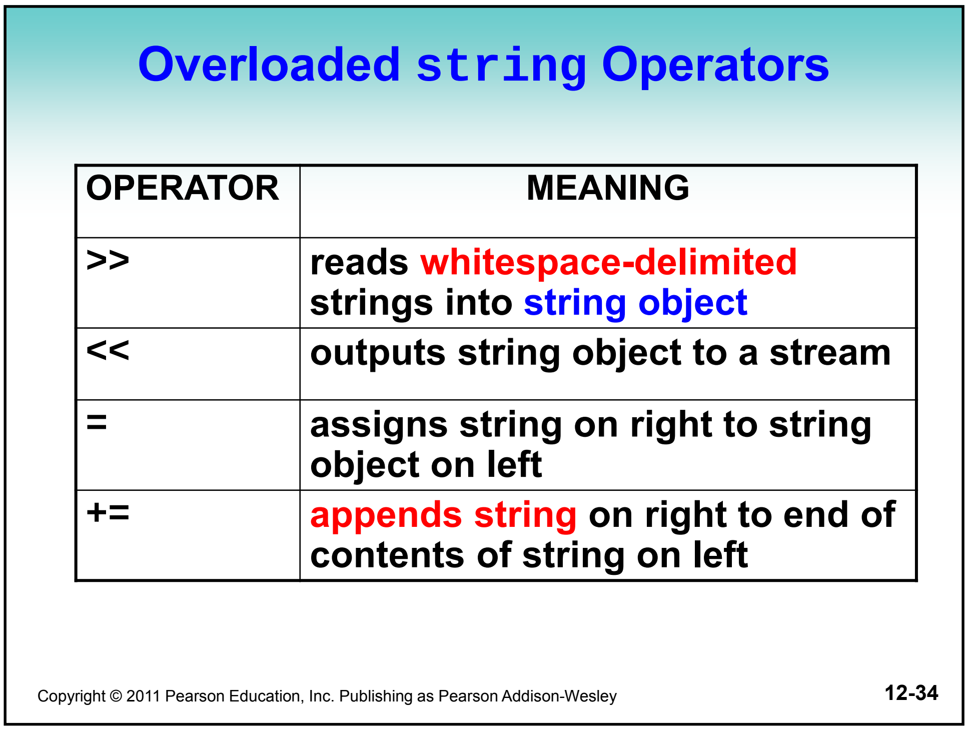
- The "string class" offers several advantages over C-style strings:
  - large body of member functions
  - overloaded operators to simplify expressions
- Need to include the `string` header file



`string` class constructors

- Default constructor `string()`
- Copy constructor `string(string&)` initializes string objects with values of other string objects
- Convert constructor `string(char *)` allows C-strings to be used wherever string class objects are expected

有了這個 Convert Constructor
必要時C-string 會自動轉態成 C++ string…



Overloaded `string` Operators

OPERATOR	MEANING
>>	reads whitespace-delimited strings into string object
<<	outputs string object to a stream
=	assigns string on right to string object on left
+=	appends string on right to end of contents of string on left

Overloaded string Operators (continued)

OPERATOR	MEANING
`+`	Returns concatenation of the two strings
`[]`	references character in string using array notation
`>, >=, <, <=, ==, !=`	relational operators for string comparison. Return `true` or `false`

Overloaded string Operators

```
string word1, phrase;
string word2 = " Dog";
cin >> word1; // user enters "Hot"
              // word1 has "Hot"
phrase = word1 + word2; // phrase has
                        // "Hot Dog"
phrase += " on a bun";
for (int i = 0; i < 16; i++)
   cout << phrase[i];  // displays
                       // "Hot Dog on a bun"
```

`string` Member Functions

Categories:

- conversion to C-strings: `c_str`, `data`
- modification: `append`, `assign`, `clear`, `copy`, `erase`, `insert`, `replace`, `swap`
- space management: `capacity`, `empty`, `length`, `resize`, `size`
- substrings: `find`, `substr`
- comparison: `compare`

Conversion to C-strings

- `data()` and `c_str()` both return the C-string equivalent of a `string` object
- Useful when using a string object with a function that is expecting a C-string

```
char greeting[20] = "Have a ";
string str(“nice day”);// 直接將C-string 轉為 C++-string
strcat(greeting, str.data());
```

Modification of `string` objects

- `str.append(string s)`
  appends contents of `s` to end of `str`
- Convert constructor for `string` allows a C-string to be passed in place of `s`

```
string str("Have a ");
str.append("nice day");
```

- `append` is overloaded for flexibility

Modification of `string` objects

- `str.insert(int pos, string s)`
  inserts `s` at position `pos` in `str`
- Convert constructor for `string` allows a C-string to be passed in place of `s`

```
string str("Have a day");
str.insert(7, "nice ");
```

- `insert` is overloaded for flexibility

str 的最後結果: “Have a nice day”

12.8 Creating Your Own String Class

- A good way to put OOP skills into practice
- The class allocates dynamic memory, so has copy constructor, destructor, and overloaded assignment
- Overloads the stream insertion (>>) and extraction operators (<<), and many other operators

Chapter 13: Advanced File and I/O Operations

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

Topics

13.1 Files

13.2 Output Formatting

13.3 Passing File Stream Objects to Functions

13.4 More Detailed Error Testing

13.5 Member Functions for Reading and Writing Files

Topics (continued)

13.6 Binary Files
13.7 Creating Records with Structures
13.8 Random-Access Files
13.9 Opening a File for Both Input and Output

13.1 Files

- A file is a set of data stored on a computer, often on a disk drive
- Programs can read from, write to files
- Used in many applications:
  - Word processing
  - Databases
  - Spreadsheets
  - Compilers

File Naming Conventions

- Different systems may have different requirements on how to name a file:
  - MS-DOS: up to 8 characters, a dot, up to a 3 character extension. No spaces. Example: `sales.dat`
- Extension often indicates purpose of file:
  - `.doc`: Microsoft Word file
  - `.cpp`: C++ source file
  - `.h`: C++ header file

Steps to Using a File

1. Open the file
2. Use (read from, write to) the file
3. Close the file

File Stream Objects

- Use of files requires file stream objects
- There are three types of file stream objects
  (1) `ifstream` objects: used for input
  (2) `ofstream` objects: used for output
  (3) `fstream` objects: used for both input and output

File Names

- File name can be a full pathname to file:
  ```
  c:\data\student.dat
  ```
  tells compiler exactly where to look
- File name can also be simple name:
  ```
  student.dat
  ```
  this must be in the same directory as the program executable, or in the compiler's default directory

Opening a File

- A file is known to the system by its name
- To use a file, a program needs to connect a suitable stream object to the file. This is known as opening the file
- Opening a file is achieved through the open member function of a file stream object

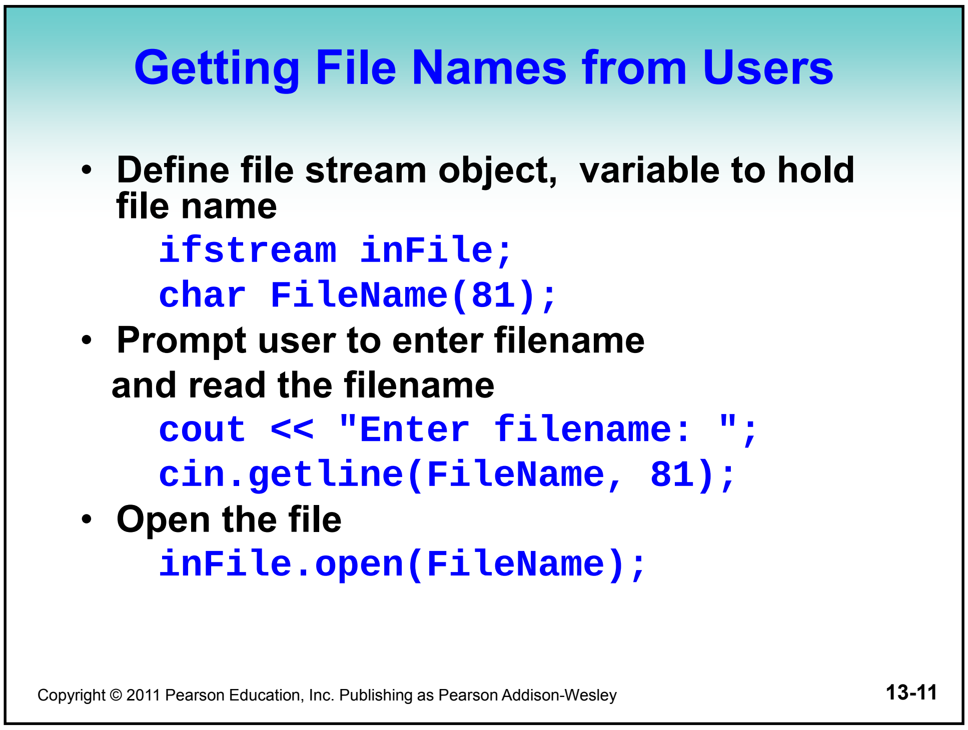
Opening a File for Input

- Create an `ifstream` object in your program

```
ifstream inFile;
```

- Open the file by passing its name to the stream's open member function

```
inFile.open("myfile.dat");
```



Getting File Names from Users

- Define file stream object, variable to hold file name

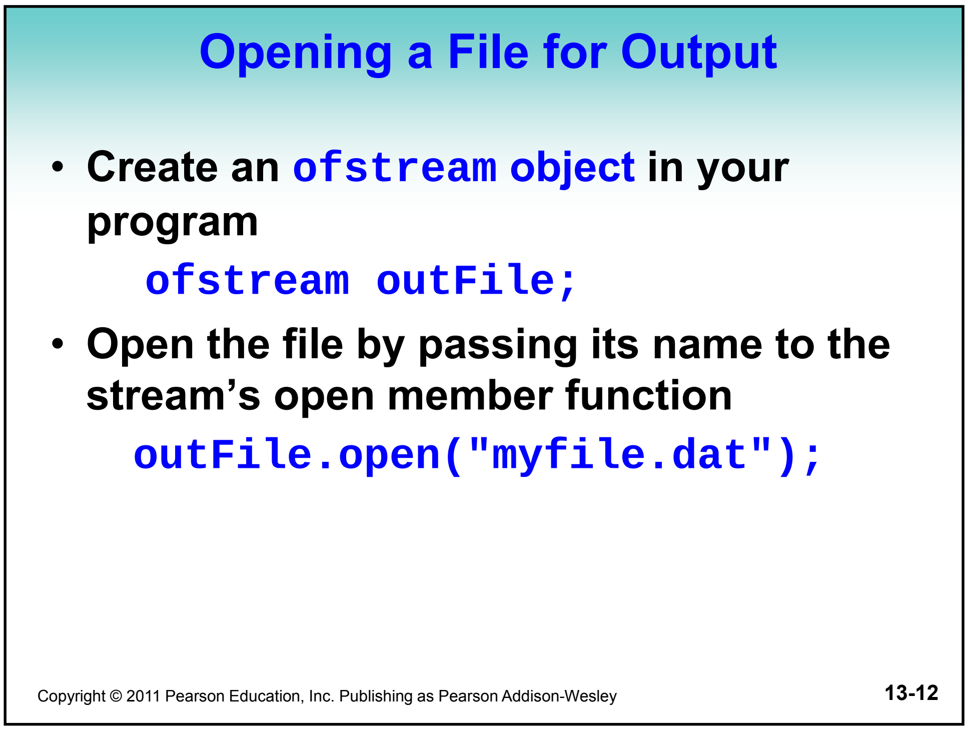
```
ifstream inFile;
char FileName(81);
```

- Prompt user to enter filename and read the filename

```
cout << "Enter filename: ";
cin.getline(FileName, 81);
```

- Open the file

```
inFile.open(FileName);
```



Opening a File for Output

- Create an `ofstream` object in your program

```
ofstream outFile;
```

- Open the file by passing its name to the stream’s open member function

```
outFile.open("myfile.dat");
```

The `fstream` Object

- `fstream` object can be used for either input or output

```
fstream file;
```

- To use `fstream` for input, specify `ios::in` as the second argument to open

```
file.open("myfile.dat",ios::in);
```

- To use `fstream` for output, specify `ios::out` as the second argument to open

```
file.open("myfile.dat",ios::out);
```

Opening a File for Input and Output

- `fstream` object can be used for both input and output at the same time
- Create the fstream object and specify both `ios::in` and `ios::out` as the second argument to the open member function

```
fstream file;
file.open("myfile.dat",
          ios::in|ios::out);
```

Opening Files with Constructors

- Stream constructors have overloaded versions that take the same parameters as open
- These constructors open the file, eliminating the need for a separate call to open

```
fstream inFile("myfile.dat",
               ios::in);
```

二合一的指令: 宣告 file stream object, 同時與檔名連結

File Open Modes

- File open modes specify how a file is opened and what can be done with the file once it is open
- `ios::in` and `ios::out` are examples of file open modes, also called file mode flag
- File modes can be combined and passed as second argument of open member function

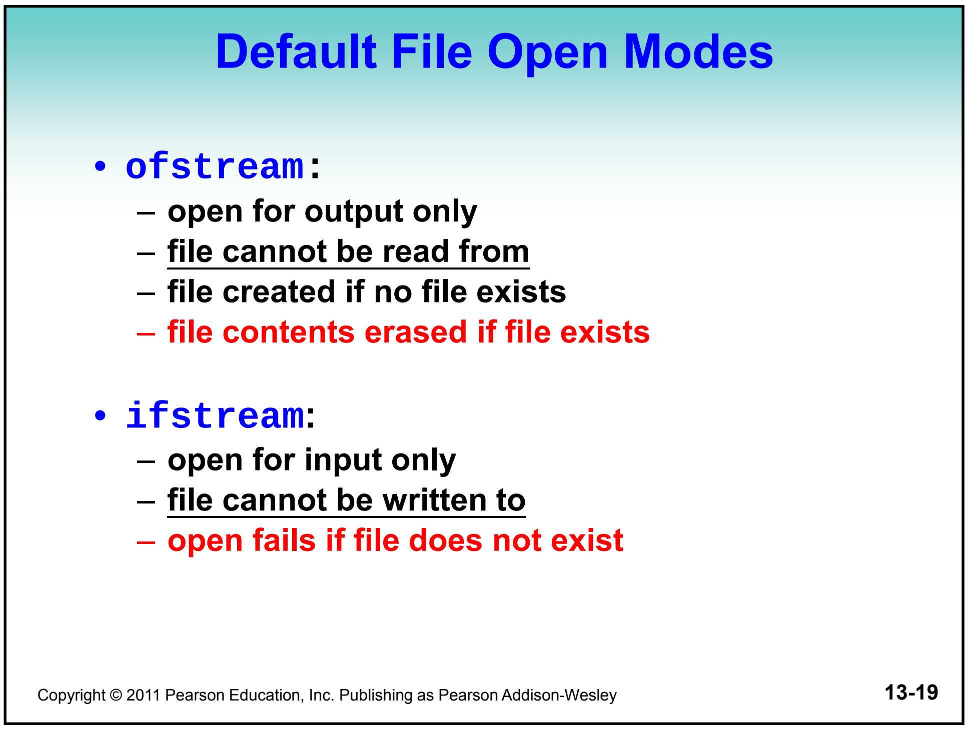
File Mode Flags

`ios::app`	create new file, or append to end of existing file
`ios::ate`	go to end of existing file; write anywhere
`ios::binary`	read/write in binary mode (not text mode)
`ios::in`	open for input
`ios::out`	open for output

"ios::ate" stands for "at end"

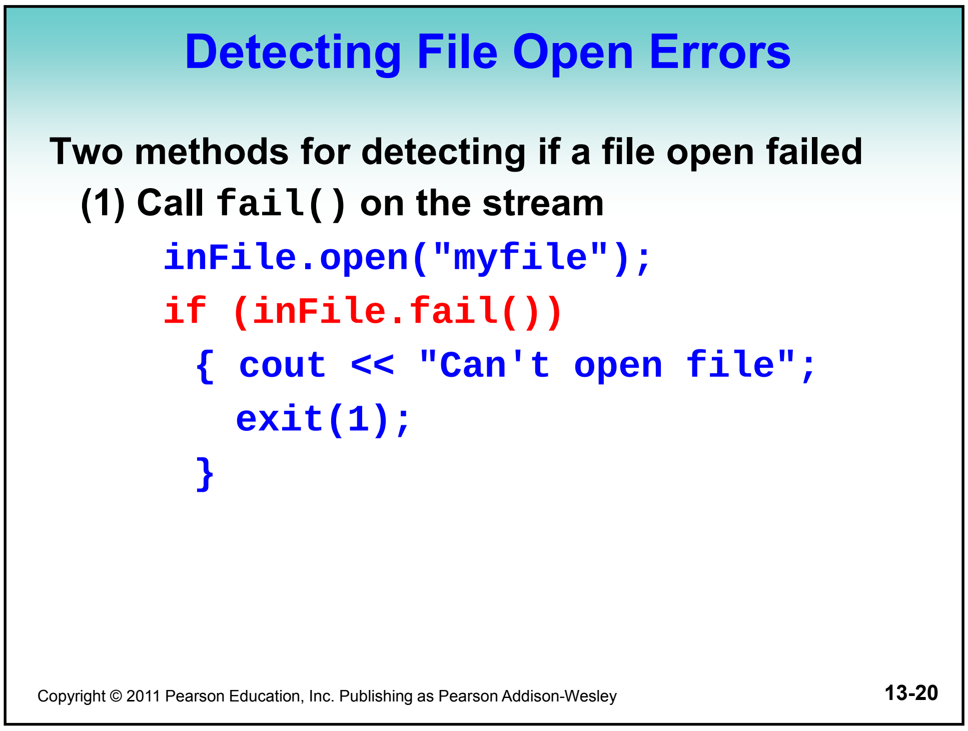
File Open Modes

- Not all combinations of file open modes make sense
- `ifstream` and `ofstream` have default file open modes defined for them, hence the second parameter to their open member function is optional



Default File Open Modes

- `ofstream`:
  - open for output only
  - file cannot be read from
  - file created if no file exists
  - file contents erased if file exists
- `ifstream`:
  - open for input only
  - file cannot be written to
  - open fails if file does not exist



Detecting File Open Errors

Two methods for detecting if a file open failed

(1) Call `fail()` on the stream

```
inFile.open("myfile");
if (inFile.fail())
  { cout << "Can't open file";
    exit(1);
  }
```

Detecting File Open Errors

(2) Test the status of the stream using the ! operator

```
inFile.open("myfile");
if (!inFile)
  { cout << "Can't open file";
    exit(1);
  }
```

比較簡潔的版本

Using `fail()` to detect eof

Example of reading all integers in a file

```
//attempt a read
int x;  infile >> x;
while (!infile.fail())
{  //success, so not eof
   cout << x;
   //read again
  infile >> x;
}
```

Using >> to detect eof

- To detect end of file, `fail()` must be called immediately after the call to >>
- The extraction operator returns the same value that will be returned by the next call to fail:
  - (`infile >> x`) is nonzero if >> succeeds
  - (`infile >> x`) is zero if >> fails

Detecting End of File

Reading all integers in a file

```
int x;
while (infile >> x)  若遇到 檔尾，則 Fail
{
   // read was successful
   cout >> x;
   // go to top of loop and
   // attempt another read
}
```

13.2 Output Formatting

- Can format with I/O manipulators: they work with file objects just like they work with cout
- Can format with formatting member functions
- The `ostringstream` class allows in-memory formatting into a string object before writing to a file

可以先在字串中格式好，再寫到檔案

I/O Manipulators

`left, right`	left or right justify output
`oct, dec, hex`	display output in octal, decimal, or hexadecimal
`endl, flush`	write newline (`endl` only) and flush output
`showpos, noshowpos`	do, do not show leading + with non-negative numbers
`showpoint, noshowpoint`	do, do not show decimal point and trailing zeroes

More I/O Manipulators

`fixed, scientific`	use fixed or scientific notation for floating-point numbers
`setw(n)`	sets minimum field output width to n
`setprecision(n)`	sets floating-point precision to n
`setfill(ch)`	uses ch as fill character

Formatting with Member Functions

- Can also use stream object member functions to format output:

```
gradeFile.width(3); // like
                    // setw(3)
```

- Names of member functions may differ from manipulators.

Formatting with Member Functions

Member Function	Manipulator or Meaning
`width(n)`	`setw(n)`
`precision(n)`	`setprecision(n)`
`setf()`	set format flags
`unsetf()`	disable format flags

`sstream` Formatting

1) To format output into an in-memory string object, include the `sstream` header file and create an `ostringstream` object

```
#include <sstream>
ostringstream outStr;
```

sstream Formatting

2) Write to the ostringstream object using I/O manipulators, or other stream member functions:

```
outStr << showpoint << fixed
       << setprecision(2)
       << 'S'<< amount;
```

sstream Formatting

3) Access the C-string inside the ostringstream object by calling its str member function

```
cout << outStr.str();
```

13.3 Passing File Stream Objects to Functions

- File stream objects *keep track of current read or write position* in the file
- When "pass a file object as parameter to a function", always uses "pass by reference"

Passing File Stream Objects to Functions

```
//print all integers in a file to screen
  void printFile(ifstream &in)
  {
    int x;
    while(in >> x){
        out << x << " ";
    }
  }
```

13.4 More Detailed Error Testing

- Streams have error bits (flags) that are set by every operation to indicate success or failure of the operation, and the status of the stream
- Stream member functions report on the settings of the flags

Error State Bits

Can examine error state bits to determine file stream status

`ios::eofbit`	set when end of file detected
`ios::failbit`	set when operation failed
`ios::hardfail`	set when an irrecoverable error occurred
`ios::badbit`	set when invalid operation attempted
`ios::goodbit`	set when no other bits are set

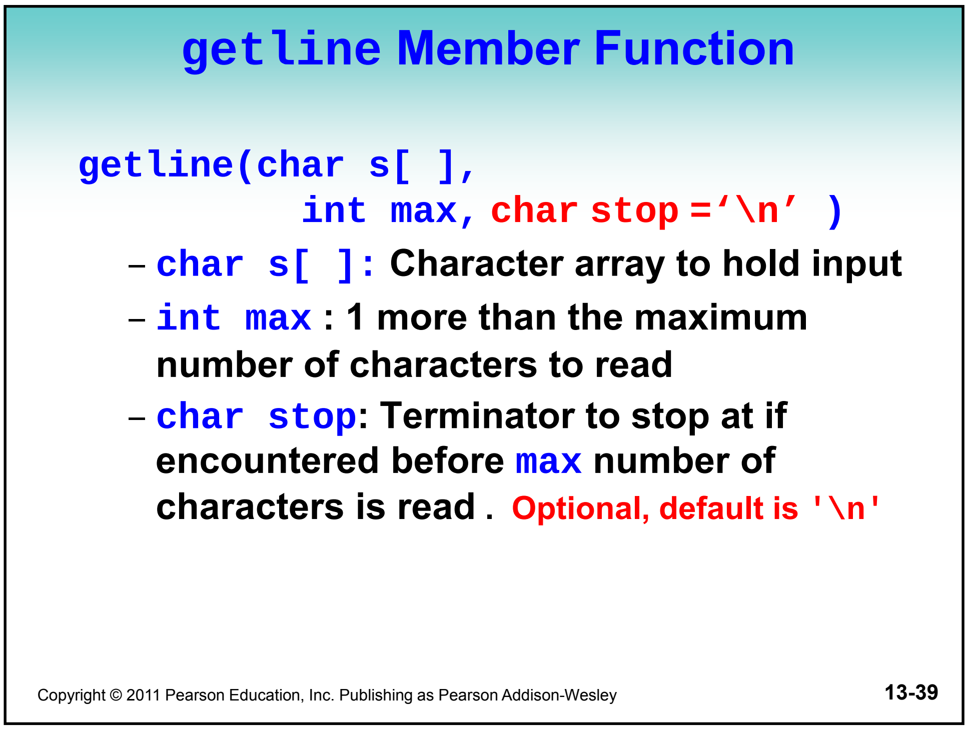
Error Bit Reporting Functions

`eof()`	true if `eofbit` set, false otherwise
`fail()`	true if `failbit` or `hardfail` set, false otherwise
`bad()`	true if `badbit` set, false otherwise
`good()`	true if `goodbit` set, false otherwise
`clear()`	clear all flags (no arguments), or clear a specific flag

13.5 Member Functions for Reading and Writing Files

- Unlike the extraction operator >>, these reading functions do not skip whitespace:
  - `getline`: read a line of input
  - `get`: read a single character
  - `seekg`: move the position in an input file
  - `seekp`: move the position in an output file

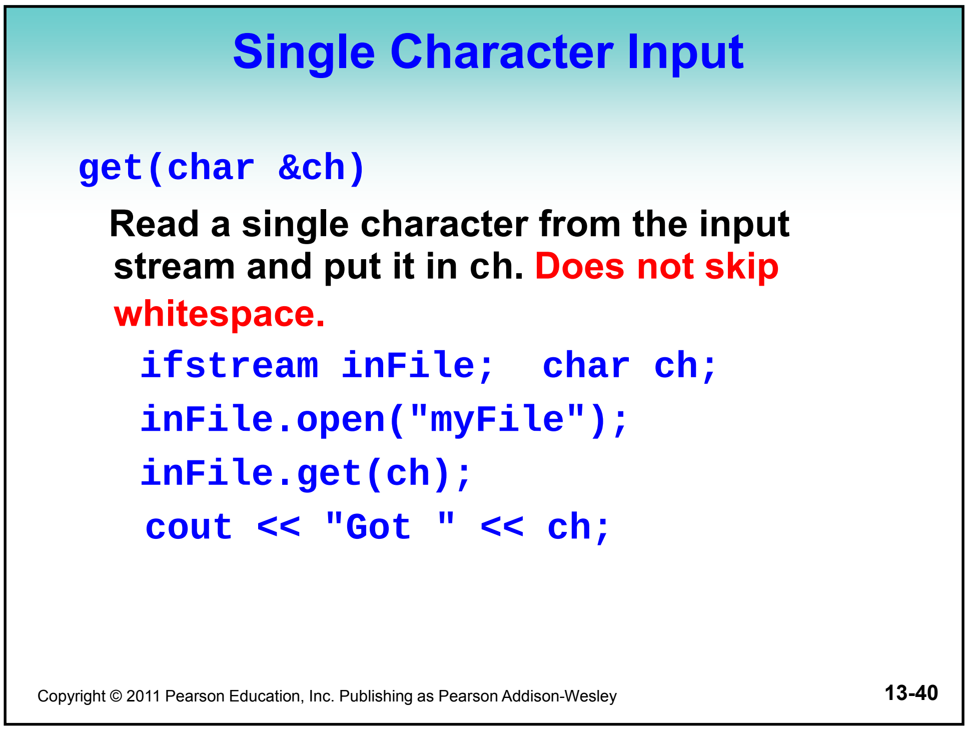
seekg() 輸入檔中移動位置, seekp()輸出檔中移動位置
seekg → "seek to get"   seekp → "seek to put"



getline Member Function

```
getline(char s[ ],
          int max, char stop ='\n' )
```

- `char s[ ]`: Character array to hold input
- `int max` : 1 more than the maximum number of characters to read
- `char stop`: Terminator to stop at if encountered before `max` number of characters is read . Optional, default is `'\n'`



Single Character Input

```
get(char &ch)
```

Read a single character from the input stream and put it in ch. Does not skip whitespace.

```
ifstream inFile;  char ch;
inFile.open("myFile");
inFile.get(ch);
cout << "Got " << ch;
```

Single Character Input, Again

`get()`

Read a single character from the input stream and return the character. Does not skip whitespace.

```
ifstream inFile;  char ch;
inFile.open("myFile");
ch = inFile.get();
cout << "Got " << ch;
```

Single Character Input, with a Difference

`peek()`

Read a single character from the input stream but do not remove the character from the input stream. Does not skip whitespace.

```
ifstream inFile;  char ch;
inFile.open("myFile");
ch = inFile.peek();
cout << "Got " << ch;
ch = inFile.peek();
cout << "Got " << ch;//same output
```

遇到檔尾，傳回 -1

Single Character Output

- `put(char ch)`

  Output a character to a file
- Example

```
ofstream outFile;
outFile.open("myfile");
outFile.put('G');
```

Single Character I/O

To copy an input file to an output file

```
char ch; infile.get(ch);
while (!infile.fail())
{
   outfile.put(ch);
   infile.get(ch);
}
infile.close();
outfile.close();
```

Moving About in an Input File

`seekg(offset, place)`

Move to a given `offset` relative to a given `place` in the file

- `offset`: number of bytes from `place`, specified as a `long`
- `place`: location in file from which to compute offset
  - `ios::beg`: beginning of file
  - `ios::end`: end of the file
  - `ios::cur`: current position in file

Rewinding a File

- To move to beginning of file, seek to an offset of zero from beginning of file

```
inFile.seekg(0L, ios::beg);
```

- Error or eof bits will block seeking to the beginning of file. Clear bits first:

```
inFile.clear();
inFile.seekg(0L, ios::beg);
```

13.6 Binary Files

- Binary files store data in the same format that a computer has in main memory
- Text files store data in which numeric values have been converted into strings of ASCII characters
- Files are opened in text mode (as text files) by default

Using Binary Files

- Pass the `ios::binary` flag to the open member function to open a file in binary mode

```
infile.open("myfile.dat",ios::binary);
```

- Reading and writing of binary files requires special `read` and `write` member functions

```
read(char *buffer, int numberBytes)
write(char *buffer, int numberBytes)
```

Using read and write

```
read(char *buffer, int numberBytes)
write(char *buffer, int numberBytes)
```

- `buffer`: holds an array of bytes to transfer between memory and the file
- `numberBytes`: the number of bytes to transfer

Address of the buffer needs to be cast to `char *` using `reinterpret_cast`

Using write

To write an array of 2 doubles to a binary file

```
ofstream
outFile("myfile", ios::binary);
double d[2] = {12.3, 34.5};
outFile.write(
   reinterpret_cast<char*>(d), sizeof(d)
);
```

d[] 原是 double *, 要先轉為 generic 的 char *

Using `read`

To read two 2 doubles from a binary file into an array

```
ifstream inFile("myfile", ios::binary);
const int DSIZE = 10;
double data[DSIZE];
inFile.read(
   reinterpret_cast<char *>(data),
   2*sizeof(double)
   );
// only data[0] and data[1] contain
// values
```

13.7 Creating Records with Structures

- Can write structures to, read structures from files
- To work with structures and files,
  - use `binary` file flag upon open
  - use `read`, `write` member functions

Creating Records with Structures

```
struct TestScore
{
  int studentId;
  float score;
  char grade;
};

TestScore test1[20];
...
// write out test1 array to a file
gradeFile.write(
   reinterpret_cast<char*>(test1),
   sizeof(test1));
```

Notes on Structures Written to Files

- Structures to be written to a file must not contain pointers
- Since string objects use pointers and dynamic memory internally, structures to be written to a file must not contain any string objects

13.8 Random-Access Files

- Sequential access: start at beginning of file and go through data in file, in order, to end
  - to access 100th entry in file, go through 99 preceding entries first
- Random access: access data in a file in any order
  - can access 100th entry directly

Random Access Member Functions

- seekg (seek get): used with input files
- seekp (seek put): used with output files

Both are used to go to a specific position in a file

Random Access Member Functions

```
seekg(offset, place)
seekp(offset, place)
```

`offset`: long integer specifying number of bytes to move

`place`: starting point for the move, specified by `ios::beg`, `ios::cur` or `ios::end`

Random-Access Member Functions

- Examples:

```
// Set read position 25 bytes
// after beginning of file
inData.seekg(25L, ios::beg);

// Set write position 10 bytes
// before current position
outData.seekp(-10L, ios::cur);
```

Random Access Information

- `tellg` member function: <u>return current byte position in input file</u>

```
int whereAmI;
whereAmI = inFile.tellg();
```

- `tellp` member function: <u>return current byte position in output file</u>

```
whereAmI = outFile.tellp();
```

13.9 Opening a File for Both Input and Output

- File can be open for input and output simultaneously
- Supports updating a file:
  - read data from file into memory
  - update data
  - write data back to file
- Use `fstream` for file object definition:

```
fstream gradeList("grades.dat",
                  ios::in | ios::out);
```

Chapter 14: Recursion

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Topics

14.1 Introduction to Recursion
14.2 The Recursive Factorial Function
14.3 The Recursive gcd Function
14.4 Solving Recursively Defined Problems
14.5 A Recursive Binary Search Function
14.6 The QuickSort Algorithm
14.7 The Towers of Hanoi
14.8 Exhaustive and Enumeration Algorithms
14.9 Recursion Versus Iteration

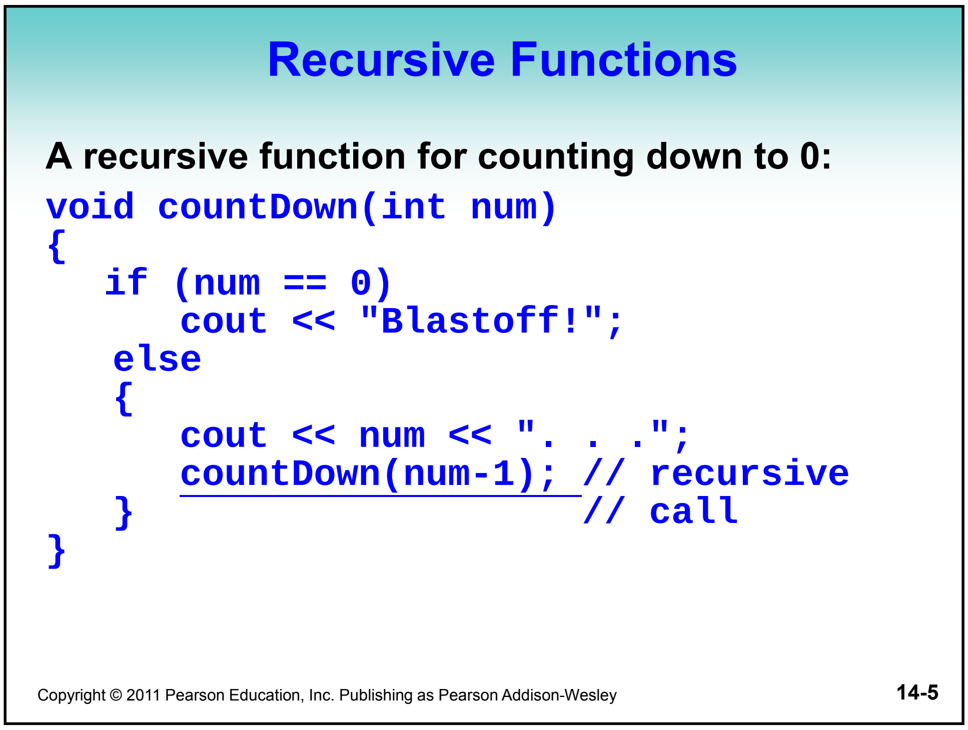
14.1 Introduction to Recursion

- A recursive function is a function that calls itself.
- Recursive functions can be useful in solving problems that can be broken down into smaller or simpler subproblems of the same type.
- A base case should eventually be reached, at which time the breaking down (recursion) will stop.

Recursive Functions

Consider a function for solving the count-down problem from some number `num` down to `0`:

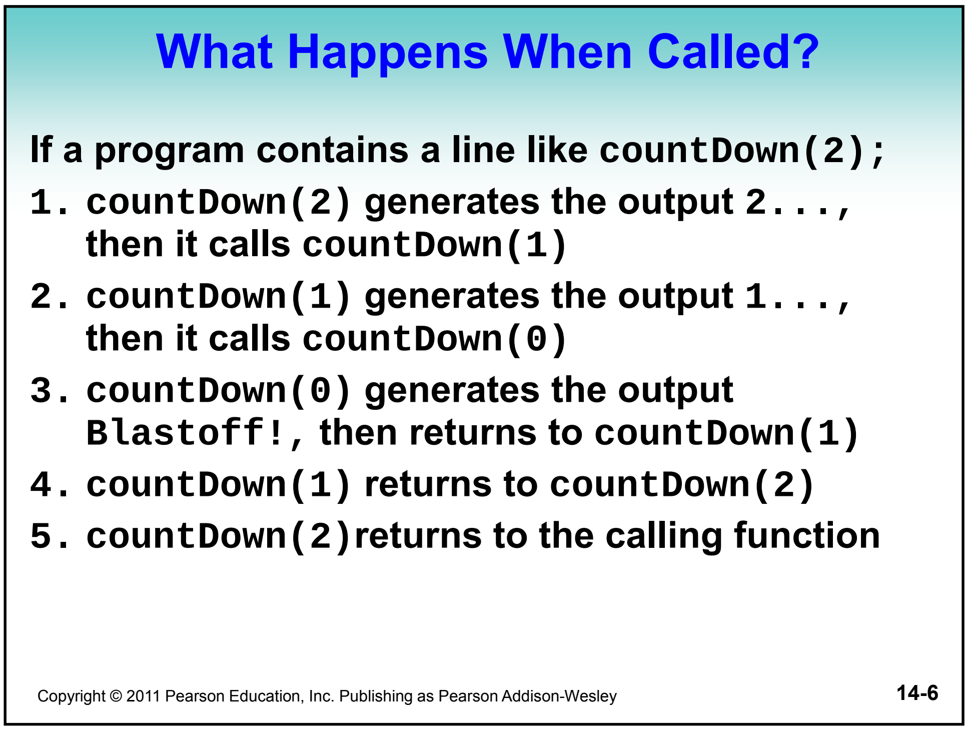
- The base case is when `num` is already `0`: the problem is solved and we "blast off!"
- If `num` is greater than `0`, we count off `num` and then recursively count down from `num-1`



Recursive Functions

A recursive function for counting down to 0:

```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << ". . .";
      countDown(num-1); // recursive
   }                    // call
}
```

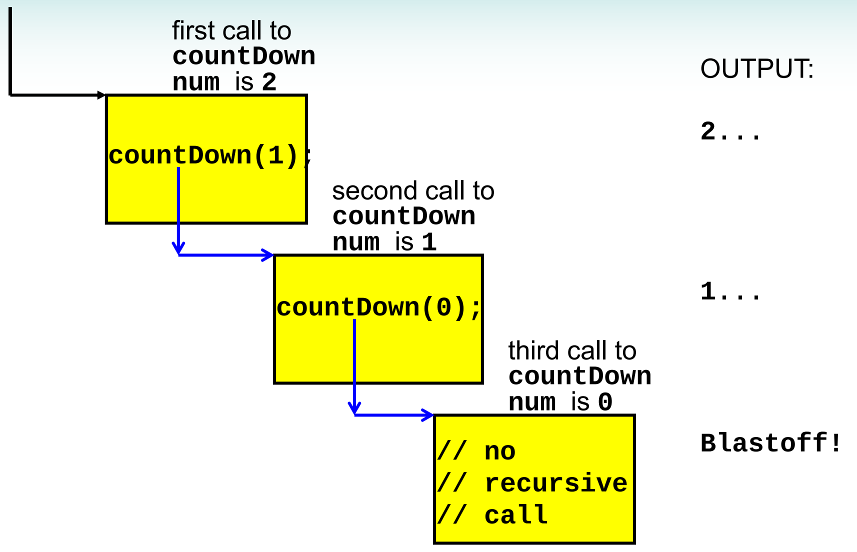


What Happens When Called?

If a program contains a line like `countDown(2);`

1. `countDown(2)` generates the output `2...,` then it calls `countDown(1)`
2. `countDown(1)` generates the output `1...,` then it calls `countDown(0)`
3. `countDown(0)` generates the output `Blastoff!`, then returns to `countDown(1)`
4. `countDown(1)` returns to `countDown(2)`
5. `countDown(2)`returns to the calling function

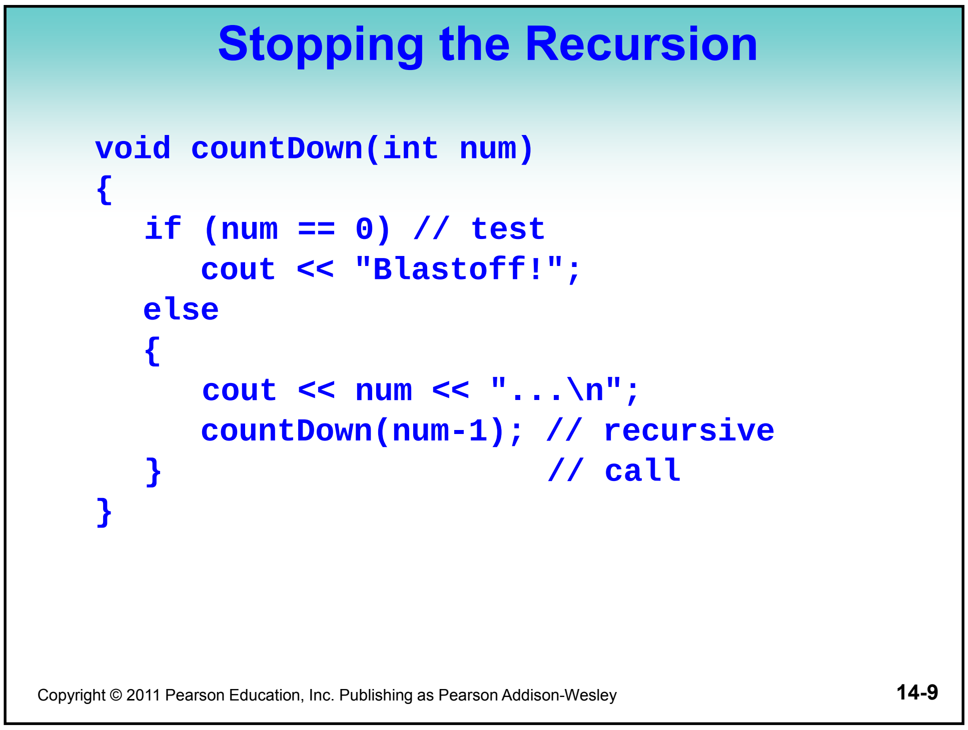
What Happens When Called?



Stopping the Recursion

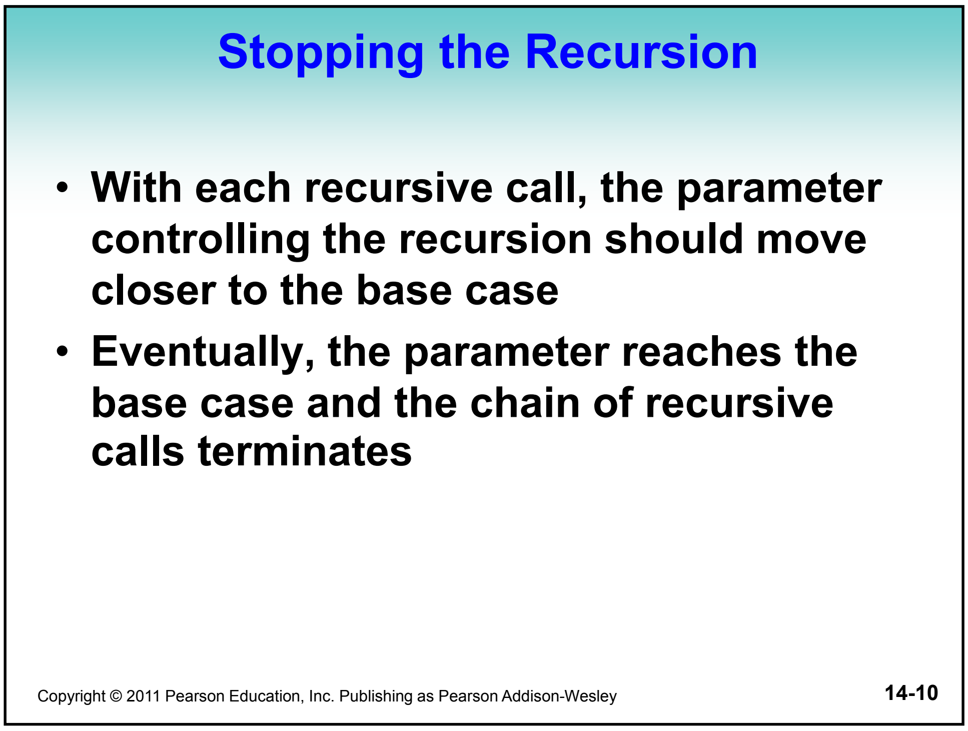
- **A recursive function should include a test for the base cases**
- **In the sample program, the test is:**

```
if (num == 0)
```



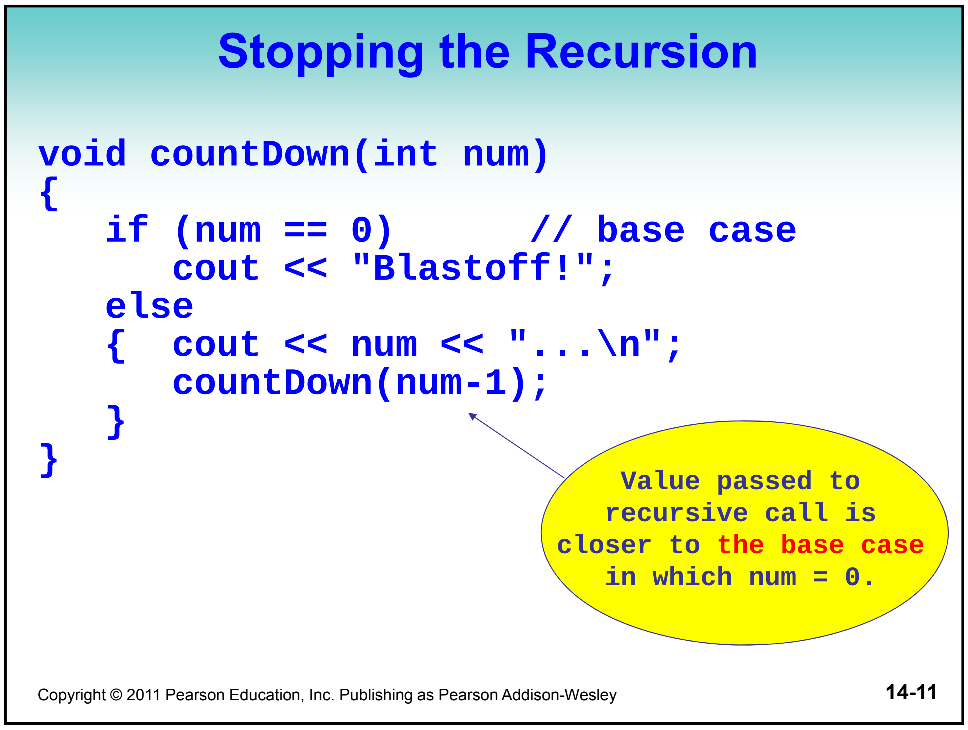
Stopping the Recursion

```
void countDown(int num)
{
   if (num == 0) // test
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive
   }                    // call
}
```



Stopping the Recursion

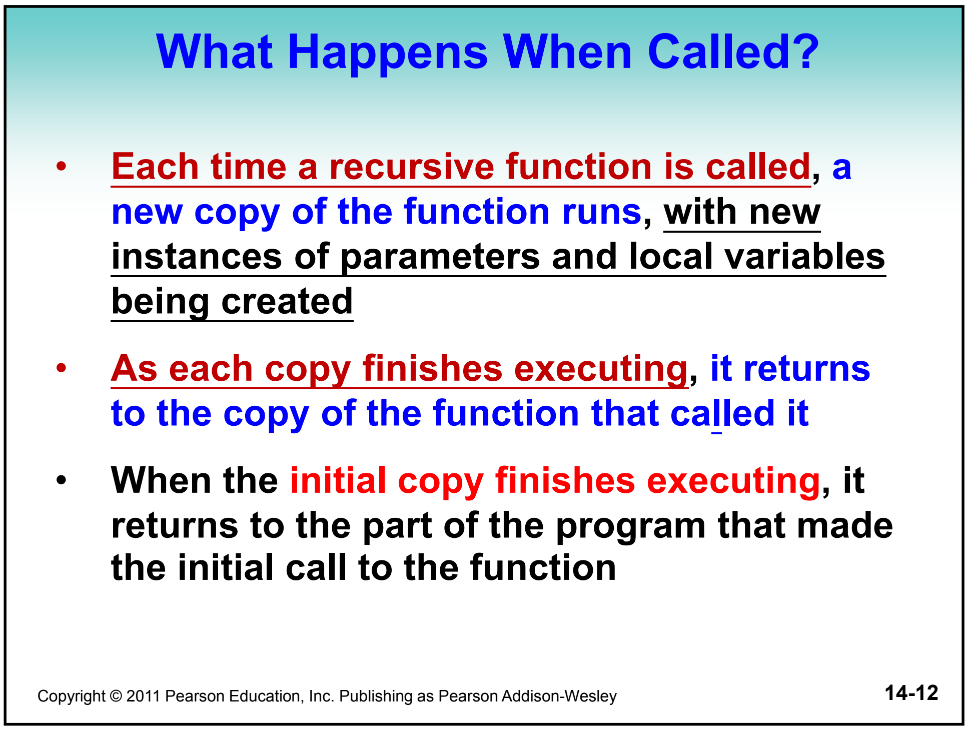
- **With each recursive call, the parameter controlling the recursion should move closer to the base case**
- **Eventually, the parameter reaches the base case and the chain of recursive calls terminates**



Stopping the Recursion

```
void countDown(int num)
{
   if (num == 0)      // base case
      cout << "Blastoff!";
   else
   {  cout << num << "...\n";
      countDown(num-1);
   }
}
```

Value passed to recursive call is closer to the base case in which num = 0.



What Happens When Called?

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables being created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

Types of Recursion

- Direct recursion
  - a function calls itself
- Indirect recursion
  - function A calls function B, and function B calls function A. Or,
  - function A calls function B, which calls …, which calls function A

14.2 The Recursive Factorial Function

- The factorial of a nonnegative integer *n* is the product of all positive integers less or equal to *n*
- Factorial of *n* is denoted by *n*!
- The factorial of 0 is 1

$$0\,! = 1$$
$$n\,! = n \times (n\text{-}1) \times \;\ldots \times 2 \times 1 \;\text{ if }\; n > 0$$

Recursive Factorial Function

- Factorial of n can be expressed in terms of the factorial of n-1

  $0\,! = 1$

  $n\,! = n \times (n\text{-}1)\,!$ Recursive Formula

- Recursive function

```
int factorial(int n)
{ if (n == 0) return 1;
  else
      return n *factorial(n-1);
}
```

14.3 The Recursive gcd Function

- Greatest common divisor (gcd) of two integers x and y is the largest number that divides both x and y
- The Greek mathematician Euclid discovered that
  - If y divides x, then gcd(x, y) is just y
  - Otherwise, the gcd(x, y) is the gcd of y and the remainder of dividing x by y

Recursive Formula

The Recursive gcd Function

```
int gcd(int x, int y)
{
    if (x % y == 0) //base case
        return y;
   else if (y % x ==0)
        return (x);
    if(x>y)
        return(gcd(y, x%y)); // x is larger
    else
        return(gcd(x, y%x)); // y is larger
}
```

14.4 Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, ...`
- After the starting `0, 1`, each term is the sum of the two preceding terms
- Recursive solution:

  `fib(n) = fib(n – 1) + fib(n – 2);`
- Base cases: `n == 0, n == 1`

Recursive Fibonacci Function

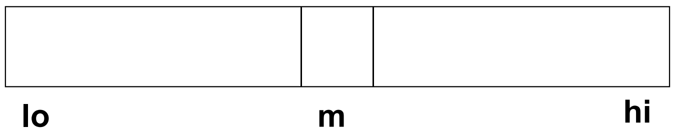
```
int fib(int n)
{
   if (n <= 0)           // base case
       return 0;
   else if (n == 1)      // base case
       return 1;
   else
         return fib(n – 1) + fib(n – 2);
}
```

14.5 A Recursive Binary Search Function

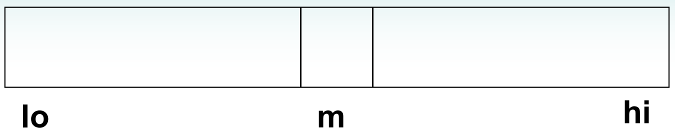
- Assume an array a that is sorted in ascending order, and an item X
- We want to write a function that searches for X within the array a, returning the index of X if it is found, and returning -1 if X is not in the array

Recursive Binary Search

A recursive strategy for searching a portion of the array from index `lo` to index `hi` is to set `m` to index of the middle portion of array:



Recursive Binary Search



If `a[m] == X,` we found `X`, so return `m`

If `a[m] > X,` recursively search `a[lo..m-1]`

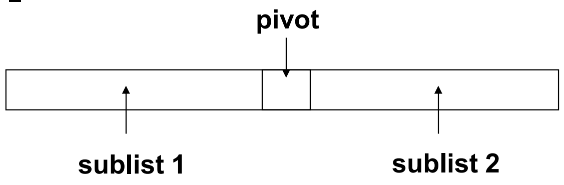
If `a[m] < X,` recursively search `a[m+1..hi]`

Recursive Binary Search

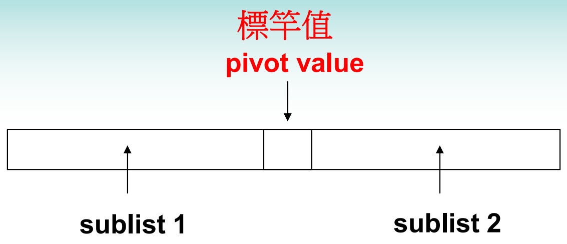
```
int bSearch(int a[],int lo,int hi,int X)
{
  int m = (lo + hi) /2;
  if(lo > hi) return -1;    // base
  if(a[m] == X) return m;   // base

  if(a[m] > X)
    return bsearch(a,lo,m-1,X);
  else
    return bsearch(a,m+1,hi,X);
}
```

14.6 The QuickSort Algorithm

- Recursive algorithm that can sort an array
- First, determine an element to use as pivot value:



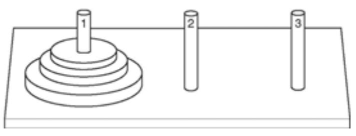
The QuickSort Algorithm



- (Basic Operation) Through a sequnece of exchanges, so that
  - (1) elements in sublist1 are < pivot
  - (2) and elements in sublist2 are >= pivot
- Recursive Calls: recursively sorts sublist1 and sublist2
- Base case: sublist has size <=1

14.7 The Towers of Hanoi

- Setup: 3 pegs, one has n disks on it, the other two pegs empty. The disks are arranged in increasing diameter, top→ bottom
- Objective: move the disks from peg 1 to peg 3, observing
  - (1) only one disk moves at a time
  - (2) all remain on pegs except the one being moved
  - (3) a larger disk cannot be placed on top of a smaller disk at any time



The Towers of Hanoi

How it works:

n=1	Move disk from peg 1 to peg 3. Done.
n=2	Move top disk from peg 1 to peg 2. Move remaining disk from peg 1 to peg 3. Move the one disk from peg 2 to peg 3. Done.

Outline of Recursive Algorithm

If n==0, do nothing (base case)

If n>0, then

a. Move the topmost n-1 disks from peg1 to peg2 (a recursive call)

b. Move the nth disk from peg1 to peg3

c. Move the n-1 disks from peg2 to peg3 (another recursive call)

end if

If complexity is denoted as $T(n)$,
Then $T(n) = 2T(n-1) + 1 \rightarrow T(n) = O(2^n)$

14.8 Exhaustive and Enumeration Algorithms

- **Enumeration algorithm**: generate **all possible combinations**

  Example: all possible ways to make change for a certain amount of money

- **Exhaustive algorithm**: search a set of combinations to **find an optimal one**

  Example: change for a certain amount of money that uses the fewest coins

14.8 Recursion vs. Iteration

- Benefits (+), disadvantages(-) for **recursion**:
  - + Natural formulation of solution to certain problems
  - + Results in shorter, simpler functions
  - – May not execute very efficiently
- Benefits (+), disadvantages(-) for **iteration**:
  - + Executes more efficiently than recursion
  - – May not be as natural as recursion for some problems

國立清華大學 電機工程學系
EE2310
Introduction to Programming

Appendix 1
Basic Programming Concepts

Outline

- **Scopes of Variables**
- **Dynamic Variable and Pointers**
- **Memory Map**
- **Call-by-Value vs Call-by-Reference**
- **Routing through a Maze**

Example C++ Program

```
#include <iostream.h>

char course_name[100] = “data structure”;      A file-scope variable

main()
{
  int a = 84;      a is a local variable
  printf(“Welcome to %s\n”, course_name);
  printf(“n is %d, n+1 is %d\n”, a, add_one(a));
}

Int add_one(int b)      b is an input argument
{
   int  c;      c is a local variable
   printf(“A subroutine for %s\n”, course_name);
   c = b + 1;
   return(c);
}
```

Example C++ Program – Global Variable

Source File 1

```
#include <iostream.h>

char course_name[100] = “data structure”;

main()
{
  int a = 84;
  printf(“Welcome to %s\n”, course_name);
  printf(“n is %d, n+1 is %d\n”, a, add_one(a));
}
```

Source File 2

```
#include <iostream.h>

extern char course_name[100] = “data structure”;

Int add_one(int b)
{
   printf(“A subroutine for %s\n”, course_name);
   return(b+1);
}
```

Global Variables

- **Problem**
  - A global variable defined in file1.C, and to be also used in file2.C
  - → Use *extern* to declare the variable in file2.C

SEGMENT A

```
#ifdef MAIN  /* macro MAIN is defined in file1.C */
   int  global_variable;
#else
   extern int global_variable /* declare extern in all other files */
#endif
```

File_1.C

```
#define MAIN
SEGMENT A
main()
{
…
}
```

File_2.C

```
SEGMENT A
subroutine_1()
{
…
}
```

Data Declaration in C++

- **(1) Constant Value**
- **(2) Variables**
- **(3) Constant Variable**
  - const int MAX = 500;
- **(4) Enumeration types**
  - enum *Boolean* {FALSE, TRUE};
- **(5) Pointers**
  - Hold memory addresses of objects
  - int *i* = 25;
  - int **np;* — *np* is a pointer to an integer, where '*' is like "taking content"
  - *np* = &*i;* — *np* points to the location of *i*, where '&' is like "taking address"

Outline

- Scopes of Variables
- Dynamic Variable and Pointers
- Memory Map
- Call-by-Value vs Call-by-Reference
- Routing through a Maze

C++ Statement and Operators

- Dynamic Memory Management
  - *"new"* and *"delete"*
- Input/Output
  - Uses shift left (<<) and shift right (>>) operators
- Operator Overloading
  - An operator could have multiple functions, depending on the types of operands that it is being applied to

Dynamic Memory Allocation

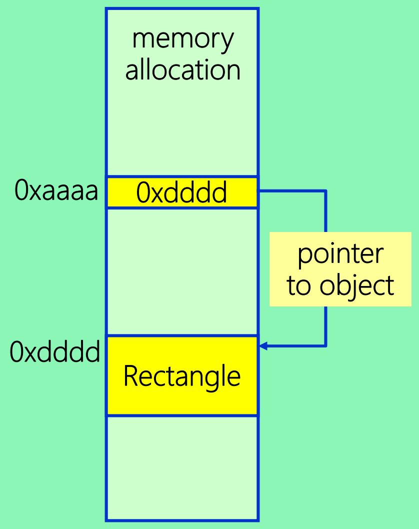
- New
  - This operator creates an object of the desired type and return a pointer to the data type that follows it.
  - It returns 0 if not being able to create it
- Delete
  - Free the data allocated by "new" operator

```
int *ip = new int;
If(ip==0) cerr << "Memory not allocated" << endl
.
.
delete ip;
```

Creating An Array

```
int *jp=new int[10];
if(jp==0) cerr << "Memory not allocated" << endl
.
.
delete [ ] jp;

/* The operator [ ] is used to inform the compiler that
the object being created or deleted is an array
```

Object vs. Pointer



```
Rectangle r, s;
Rectangle *t = &s;
```

symbol table (used in Compiler)

name	type	address
s	Rectangle	0xdddd
t	pointer	0xaaaa

Data Declaration in C++ (con't)

- **(6) Reference types**
  - **A unique feature of C++, (which is not available in C)**
  - **Is a mechanism to provide an alternative name for an object**
  - **Example**

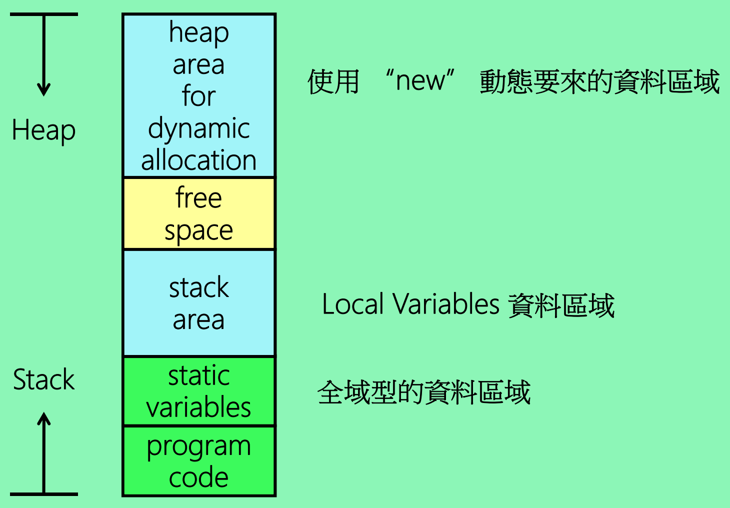
```
int i=5;
int& j=i;
i=7;
printf("i=%d, j=%d", i, j); → both i and j are 7;
```

Outline

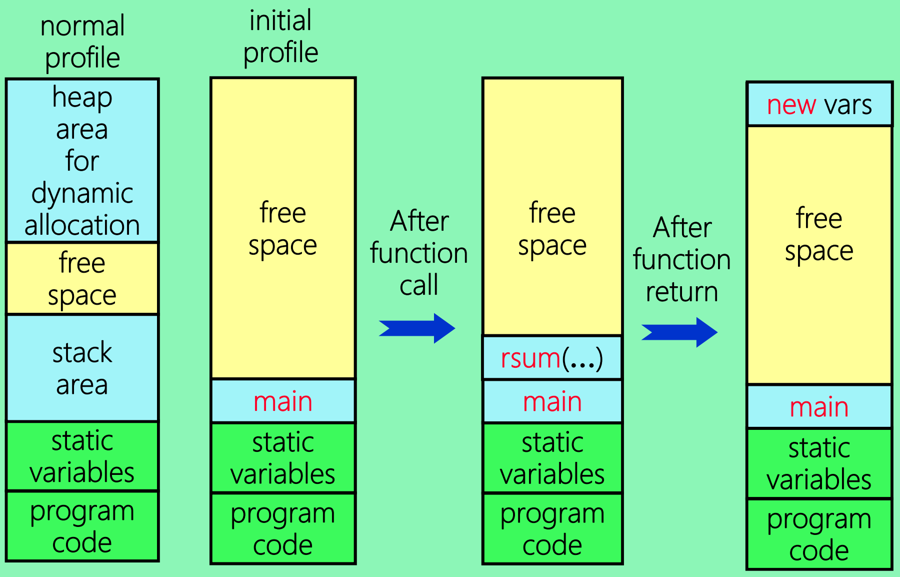
- Scopes of Variables
- Dynamic Variable and Pointers
- **Memory Map**
- Call-by-Value vs Call-by-Reference
- Routing through a Maze

Memory Allocation

Normal profile



Memory Allocation – Subroutine Invocation



Outline

- **Scopes of Variables**
- **Dynamic Variable and Pointers**
- **Memory Map**
- **Call-by-Value vs Call-by-Reference**
- **Routing through a Maze**

Parameter Passing in C++

- (1) Pass by value (傳值呼叫)
  - Default mechanism
  - When an object is passed by value → it is copied into the function's local storage
  - could be slow when data to be passed is large !
- (2) Pass by reference (傳地址呼叫)
  - Done by appending and & to its type specifier
  - E.g., int max(int& a, int& b);
  - When an object is passed by reference → only the address of its location is copied into the function's local store
  - faster but less secure !

Call By Value Example

```
main()
{
   int i, j;
   cout << "Input 2 numbers:" << endl;
   cin >> i >>j;
   if( i > j )
                  swap(&i, &j);
                  cout << "The smaller number is " << i << endl;
                  cout << "The larger is " << j << endl;
};

void swap(int *ptr_x, int *ptr_y)  // call by pointer
{
    int temp;
    temp = *ptr_x;
    *ptr_x = *ptr_y;
    *ptr_y = temp;
}
```

Call By Reference Example

```
main()
{
   int i, j;
   cout << "Input 2 numbers:" << endl;
   cin >> i >>j;
   if( i > j )
                swap(i, j);
                cout << "The smaller number is " << i << endl;
                cout << "The larger is " << j << endl;
};

void swap(int &x, int &y)  // call by reference
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

主程式好像是傳值

副程式卻能看到主程式內的變數，直接存取

Pass by Const References

主程式只開放副程式
看得到所傳的 變數或物件，但不能更改

- **A Best Method**
  - **pass by "const T& *a*", T is the type of the argument *a***
  - **Faster than pass-by-value if a large chunk of arguments to be passed**
  - **Better protection of the actual arguments to be passed**
  - **Any attempt to modify a *const* argument in the function body will result in a compile-time error**

Improper manipulations of the input arguments
→ could lead to nasty bugs

Pass-by-Value vs. Pass-by-Reference

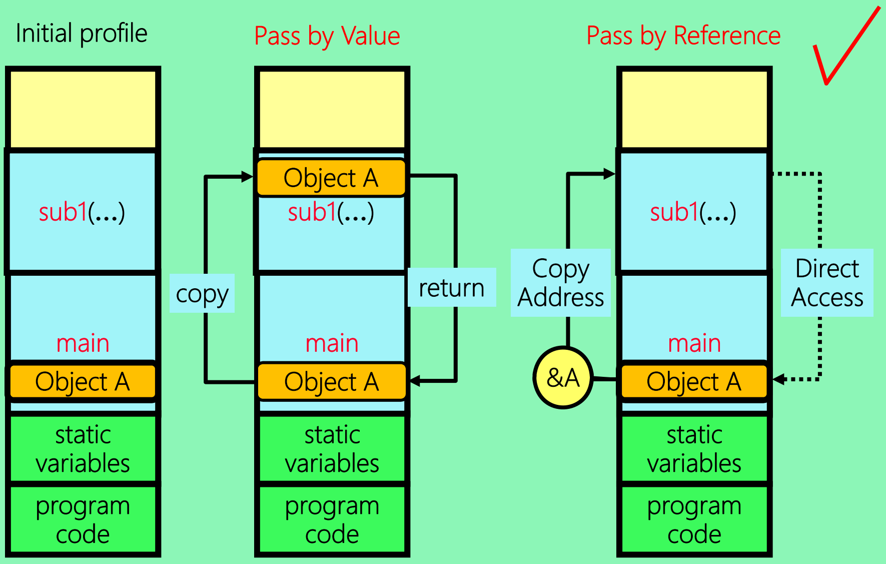
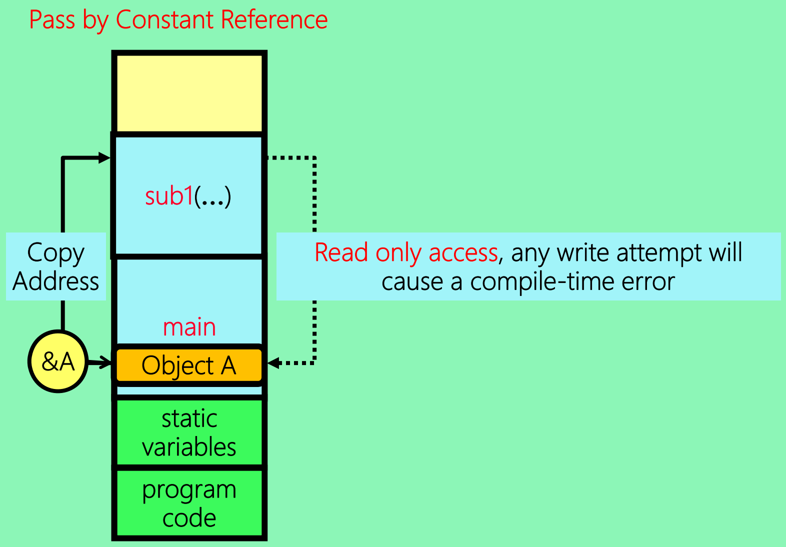


Illustration: Pass by Const References



Pass by Constant Reference
sub1(...)
main
Object A
static variables
program code
Copy Address
&A
Read only access, any write attempt will cause a compile-time error

One Exception

- **Array**
  - Does not pass by value
  - I.e., it is not copied to the function's local store
  - Only the pointer of the first element is passed
  - Function is not aware of the size of the array
  - Often the size of an array is also passed as another argument

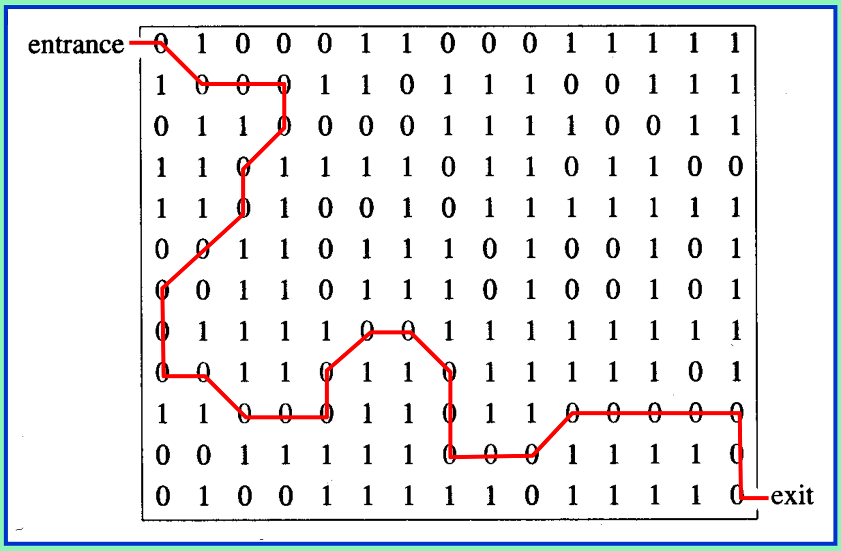
例子: A subroutine that sorts an array of *n* integer elements
Subroutine 結構如下:

```
float sorting(float *a, const int n) {
    // where a is the array name
    …..
}
```

Outline

- Scopes of Variables
- Dynamic Variable and Pointers
- Memory Map
- Call-by-Value vs Call-by-Reference
- ➡ Routing through a Maze

An Example Maze



```
0 1 0 0 0 1 1 0 0 0 1 1 1 1 1
1 0 0 0 1 1 0 1 1 1 0 0 1 1 1
0 1 1 0 0 0 0 1 1 1 1 0 0 1 1
1 1 0 1 1 1 1 0 1 1 0 1 1 0 0
1 1 0 1 0 0 1 0 1 1 1 1 1 1 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 1 1 1 1 0 0 1 1 1 1 1 1 1 1
0 0 1 1 0 1 1 0 1 1 1 1 1 0 1
1 1 0 0 0 1 1 0 1 1 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0 1 1 1 1 0
0 1 0 0 1 1 1 1 1 0 1 1 1 1 0
```

Problem: Routing in a Maze

Write a C++ program to find **routing path from the source point (marked as 'S') to the destination point (marked as 'F') in a maze**. Note that the maze is 10 rows by 20 columns, with a '*' denoting a "blockage square" and a '-' denoting a "passage square". This maze is generated by a code segment as in a template cpp file, called "**template.hw6.cpp**". It is highly suggested that you use this file as a template to finish your program. Display the results as shown below. A symbol of 'P' means a routing square. (Hint: diagonal moves are allowed as illustrated in the figure).

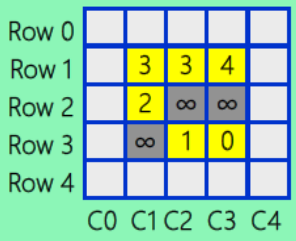
Original Maze

```
S*--*-**--**-*--*-**
-------*****-**-***-
---------**--***--**
**--**---***-*--*--*
*---*-**-*-**-***-*-
-***--**-*-*---**-*-
*--****-------*-*-**
**--****-*-**--*---*
-*-**-*----*--*-***-
---***-****-*-*--*-F
```

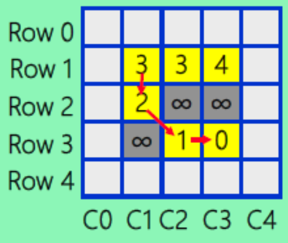
A Routing Path from 'S' to 'F'

```
S*--*-**--**-*--*-**
-PPPPP-*****-**-***-
------P--**--***--**
**--**-P-***-*--*--*
*---*-**P*-**-***-*-
-***--**P*-*---**-*-
*--****--PPPPP*-*-**
**--****-*-**-P*PPP*
-*-**-*----*--*P***P
---***-****-*-*--*-F
```

Routing Through a Maze



after step1



after step2

```
Backward_search_algorithm
{
  Step 1:
      compute the distance to the destination
       of each node by wave-front propagation
  Step 2:
     find a shortest path from the source to
     the destination node by picking up the
     nodes with a shortest distance
}
```