The Payroll Program

The following example is a simple payroll program that illustrates most of the core elements of the C++ language covered in sections 3 through 6 of the course notes. During the term, a formal specification of the program will be available via the course Website:

`ei.cs.vt.edu/~cs1044/`

Sample input and output files:

```
3175 45 7597.43 B McCoy, Leonard
5328 47 3000.00 D Checkov, Pavel
3236 35 3897.89 B Sulu, Hikaru
2334 49 2089.10 D Riley, Kevin
5315 73 9336.00 B Kirk, James
7641 58 5409.12 D Uhura
7077 23 1001.00 B Rand, Janice
4098 32 8594.98 D Spock
3424 39 2999.99 B Solo, Han
4297 52 1000.00 B Chewbacca
1716 53 2011.50 D Skywalker, Luke
5956 67 5245.42 D Scott, Montgomery
```

```
Programmer:  Bill McQuain
StarFleet Payroll

   ID    Gross Pay     FIT       SSI       Ins       Net Pay
============================================================
  3175     7597.43   2227.15    518.90    100.00     4751.37
  5328     3000.00    710.00    204.90    180.00     1905.10
  3236     3897.89   1006.30    266.23     70.00     2555.36
  2334     2089.10    454.95    142.69    180.00     1311.47
  5315     9336.00   2800.88    637.65    150.00     5747.47
  7641     5409.12   1505.01    369.44    180.00     3354.67
  7077     1001.00    150.28     68.37     70.00      712.35
  4098     8594.98   2556.34    587.04    140.00     5311.60
  3424     2999.99    710.00    204.90    100.00     1985.09
  4297     1000.00    150.00     68.30    100.00      681.70
  1716     2011.50    433.22    137.39    180.00     1260.89
  5956     5245.42   1450.99    358.26    250.00     3186.17
============================================================
  Avg:     4348.54   1179.59    297.00    141.67     2730.27
```


Simple Version

```
////////////////////////////////////////////////////////////////////
// Identification
//
//  Title:          Simple Payroll Program
//  Programmer:     Bill McQuain
//  ID Number:      999-99-9999
//  Compiler:       Visual C++ version 6.0
//  Platform:       Pentium II 400 / Windows NT
//  Last Modified: January 2, 1999
//
////////////////////////////////////////////////////////////////////
// Purpose of the program:
//
//  This program reads information about an arbitrary number of
//  employees from an input file named "workers.dat":
//
//    - ID number
//    - age
//    - gross monthly salary
//    - insurance plan type
//
//  and computes the correct deductions for:
//
//    - federal income tax
//    - social security tax
//    - insurance fee
//    - net monthly salary.
//
//  It then prints out a labeled table of results, showing for
//  each employee the ID number, gross salary, insurance fee,
//  income and social security tax deductions and net salary.
//  In addition, the averages of each category are computed and
//  printed.  Output is is to a file named "payola.dat".
//
```


Simple Version

```
//  The is the #include section.  These statements tell the compiler
//  to read and use variables and functions declared in the
//  specified files.
#include <fstream.h>
#include <iomanip.h>

//  Every C++ program must have a function named main.  This is the
//  beginning of the definition of the main function for this
//  simple program.  In this case the whole program consists of just
//  the main function --- usually programs involve more than one
//  function.
//
int main() {
//  In this section, named constants are declared and given
//  their values.  We could just use the numbers themselves in the
//  code that follows, but the names make the code more readable;
//  the name SSRate carries meaning where the number 0.0683 doesn't.
//
  const double HiTax  = 0.33;           // income tax rates
  const double MidTax = 0.28;
  const double LoTax  = 0.15;
  const double SSRate = 0.0683;         // ssi tax rate

  const double HiBasic   = 150.00;      // insurance fees
  const double MidBasic  = 100.00;
  const double LoBasic   =  70.00;
  const double HiDeluxe  = 250.00;
  const double MidDeluxe = 180.00;
  const double LoDeluxe  = 140.00;

  const double HiIncome  = 3000.00;     // income levels
  const double MidIncome = 1000.00;

  const char Basic  = 'B';              // insurance plan type codes
  const char Deluxe = 'D';
```


Simple Version

```
//  In this section, the variables used in main() are declared.
//  Each variable has a type (integer, double, etc) but no particular
//  value at this point.  The variable names like the constant names
//  used above are supposed to be meaningful, and each declared
//  variable is also given a short comment explaining what it will
//  be used for.
//
  ifstream inPay;          // input file stream
  ofstream outPay;         // output file stream

  int    IdNum,            // employee ID number,
         Age,              //          age.
         NumEmployees;     // number of employees.
  double GrossPay,         // employee gross pay.
         NetPay,           //          net pay,
         SSI,              //          SS tax,
         FIT,              //          income tax,
         InsFee,           //          insurance fee..
         TotalGross,       // total of all gross pay,
         TotalNet,         //          net pay,
         TotalIns,         //          insurance fees,
         TotalSSI,         //          SSI tax,
         TotalFIT;         //          income tax.
  char   InsPlan;          // employee insurance plan
```


Simple Version

```
//  This is the initialization section of main().  We must ensure
//  every variable is given a value before it is used in a
//  calculation or printed.  Some variables get their values by being
//  read from the input file, so they don't need to be initialized
//  here.
  NumEmployees =   0;      // Initialize counters and running totals
  TotalGross   = 0.0;      // to zero.
  TotalNet     = 0.0;
  TotalIns     = 0.0;
  TotalSSI     = 0.0;
  TotalFIT     = 0.0;

//  The following four statements prepare the input and output files
//  for use.  Don't worry too much about what's going on here just
//  yet, we'll cover the details shortly.
//
  inPay.open("workers.dat");       // open input and output files
  outPay.open("payola.dat");
  outPay.setf(ios::fixed, ios::floatfield);
  outPay.setf(ios::showpoint);

//  The following statments print the header info to output file.
//  It's rather cryptic at this point, but you'll see output is
//  really very  simple in C++.

  outPay << "Programmer:  Bill McQuain" << endl;
  outPay << "StarFleet Payroll" << endl << endl;
  outPay << "   ID    Gross Pay     FIT       SSI       Ins"
         << "       Net Pay";
  outPay << endl;
  outPay << "=============================================="
         << "==============";
  outPay << endl;
```


Simple Version

```
//  This statement tries to read first line of data from input file.
//  If something goes wrong, the variable inPay will be false (small
//  lie) and  that will prevent us from going into the loop that
//  follows and generating nonsense.

  inPay >> IdNum >> Age >> GrossPay >> InsPlan;

//  What comes next is a while loop.  The body of the loop gets
//  executed over and over again until the expression that follows
//  the while (inPay in this case) becomes false.  Therefore, we'd
//  better be sure that inPay will eventually become false,
//  otherwise the program will continue to execute the body of this
//  loop forever.

  while (inPay) {

       //  The first part of the loop body (down to the comment with
       //  all the hyphens) processes the data that was just read in.

       NumEmployees++;            // Count employees.  This statement
                                  // adds 1 to the value of the
                                  // variable NumEmployees.
       TotalGross += GrossPay;    // Update total gross pay.  This
                                  // statement adds the value of
                                  // the variable GrossPay to the
                                  // variable TotalGross
```


Simple Version

```
//  The switch statement tries to match the value of the
//  selection variable (InsPlan this time) with the given cases.
//  When a match is found, the statements that accompany that
//  case are executed.  If no match is found, the default case
//  is carried out.

switch (InsPlan) {                // Calculate insurance fees:
case Basic:{ if (Age <= 35)       // for Basic plan
               InsFee = LoBasic;  // This statement copies the
                                  // value of the constant
                                  // LoBasic to the variable
                                  // InsFee -- assignment.
             else if (Age <= 65)
               InsFee = MidBasic;
             else
               InsFee = HiBasic;
             break;
           }
case Deluxe:{ if (Age <= 35)      // for Deluxe plan
                InsFee = LoDeluxe;
              else if (Age <= 65)
                InsFee = MidDeluxe;
              else
                InsFee = HiDeluxe;
              break;
            }
default :{ cout << "Employee " << setw(4) << IdNum;
           cout << " has invalid insurance plan." << endl;
           InsFee = 0.0;
         }
}
```



```
      TotalIns += InsFee;            // Update total insurance fees.

    if (GrossPay <= MidIncome) {    // Determine FIT amount.
       FIT = LoTax * GrossPay;             // low income
    }
    else if (GrossPay <= HiIncome) {
       FIT = LoTax * MidIncome             // middle income
           + MidTax * (GrossPay - MidIncome);
    }
    else {
       FIT = LoTax * MidIncome             // high income
           + MidTax * (HiIncome - MidIncome)
           + HiTax * (GrossPay - HiIncome);
    }
    TotalFIT += FIT;               // Update total income taxes.

    SSI = SSRate * GrossPay;       // Calculate SSI tax.
    TotalSSI += SSI;               // Update total SS taxes.

    NetPay = GrossPay - InsFee -
                              FIT - SSI;  // Calculate net pay.
    TotalNet += NetPay;                   // Update total net pay.

    //  This section of the loop body prints the results calculated
    //  above into the output file.

    outPay << setw( 6) << IdNum;
    outPay << setw(12) << setprecision(2) << GrossPay;
    outPay << setw(10) << setprecision(2) << FIT;
    outPay << setw(10) << setprecision(2) << SSI;
    outPay << setw(10) << setprecision(2) << InsFee;
    outPay << setw(12) << setprecision(2) << NetPay;
    outPay << endl;

    // -------------------------------- End of Processing Section
```



```
    //  This statement skips over the employee names to the
    //  beginning of the next line of input.
    inPay.ignore(200, '\n');

    //  This statenent tries to read the next line of input.  This
    //  is the last statement in the loop body, so the next thing
    //  that will happen is that the loop control expression (inPay
    //  in this case) is tested.
    inPay >> IdNum >> Age >> GrossPay >> InsPlan;
  }                                          // End of while loop.

//  Once we've exited the loop, we're done reading and processing
//  data for new employees.  It's time to sum things up.  The
//  following statements print out the averages in a nice form.
//  First print a line to mark the end of the table body:

  outPay << "=========================================="
         << "================";
  outPay << endl;

//  Now, calculate and print averages, if appropriate.  We've got to
//  be careful not to calculate averages if there weren't any
//  employees, so this uses an if statement to be sure that's not
//  the case before any dividing is done.

  if (NumEmployees > 0) {
    outPay << "  Avg:";
    outPay << setprecision(2);
    outPay << setw(12) << TotalGross/NumEmployees;
    outPay << setw(10) << TotalFIT/NumEmployees;
    outPay << setw(10) << TotalSSI/NumEmployees;
    outPay << setw(10) << TotalIns/NumEmployees;
    outPay << setw(12) << TotalNet/NumEmployees;
    outPay << endl;
  }
```



```
//  These two statements just close the input and output files; this
//  tells the operating system that we're done with them and
//  (hopefully) that the files are properly saved to disk.

  inPay.close();
  outPay.close();

//  The return statement sends a value back to whoever called this
//  function.  If you're using Visual C++, you may see this value
//  displayed in the window at the bottom of the screen when you run
//  the program.

  return NumEmployees;
}
```


The Variations

The slides that follow show several variations and extensions of the simple payroll program. These variations illustrate the use of many of the language features covered in sections 6 through 12 of the course notes. In short:

FuncPay.cpp	modularization of Payroll.cpp using user-defined functions
ArrayPay.cpp	improved data handling using parallel arrays to store employee data
SortPay.cpp	output results sorted by net pay
StructPay.cpp	improved data organization using a single array of struct variables to store all employee data
NamesPayA.cpp	input and output of character strings; use of multiple input files; use of char array for strings and of some of the standard library string functions
NamesPayS.cpp	modification of NamesPayA.cpp to use standard library string objects

During the term, the source for these examples will be available on the course website:

`http://ei.cs.vt.edu/~cs1044/`


```
////////////////////////////////////////////////////////////////
//  Title:               Simple Payroll with Functions
//  Programmer:     Bill McQuain
//  ID Number:      999-99-9999
//  Compiler:       Microsoft Visual C++ ver 6.0
//  Platform:       Pentium II 400, Windows NT
//  Last modified:  January 2, 1999
//
//  Purpose:
//
//    The same as the simple payroll program, Pay99.cpp.
//
//  Implementation Issues:
//
//    This is an alteration of Pay98.cpp to incorporate user-
//    defined functions and an improved modular design.
//
//    While there are no changes to the calculation logic,
//    the organization of the code is substantially different.
//
//    Extensive comments describe the changes made to the
//    original.
//
#include <fstream.h>
#include <iomanip.h>
```



```
//  Beginning of main function:int main() {
//  Function prototypes.  It's not necessary to include names for
//  the parameters here, but I recommend it because it helps keep
//  things straight.

double FedTax(double Gross);
double Insurance(char Plan, int Age);
void   PrintHeader(ofstream& outPay);
void   PrintLine(ofstream& outPay, int IdNum, double Gross,
                  double Fee, double FIT, double SSI, double Net);
void   PrintAvg(ofstream& outPay, double TotalGross, double TotalIns,
                double TotalFIT, double TotalSSI, double TotalNet,
                int NumEmployees);

//  Named constants --- the use of named constants is required for
//  this program.  Other named are used in the functions that find
//  the income tax and insurance amounts.
//
const double SSRate = 0.0683;   // SSI tax rate

//  Variables:
  ifstream inPay;       // input file stream
  ofstream outPay;      // output file stream

  int    IdNum,         // employee ID number
         Age,           // employee age
         NumEmployees;  // number of employees
  double GrossPay,      // employee gross pay
         NetPay,        // employee net pay
         SSI,           // employee SS tax
         FIT,           // employee income tax
         InsFee,        // employee insurance fee
// declarations continued on next slide
```



```
          TotalGross,    // total of all gross pay
          TotalNet,      // total of all net pay
          TotalIns,      // total of all insurance fees
          TotalSSI,      // total of all SS tax
          TotalFIT;      // total of all income tax
  char    InsPlan;       // employee insurance plan

// Initializations:  NumEmployees =   0;
// Initialize counters and running totals
  TotalGross   = 0.0;    // to zero.
  TotalNet     = 0.0;
  TotalIns     = 0.0;
  TotalSSI     = 0.0;
  TotalFIT     = 0.0;

  inPay.open("workers.dat");       // open input and output files
  outPay.open("payola.dat");
                              // Set up for floating point output:
  outPay.setf(ios::fixed, ios::floatfield);
  outPay.setf(ios::showpoint);
// Print header info to output file:
  PrintHeader(outPay);
// Read first line of data from input file:
  inPay >> IdNum >> Age >> GrossPay >> InsPlan;
```



```
// Process and read next line until there's no more data:
  while (inPay) {

     NumEmployees++;                    // Count employees.
     TotalGross += GrossPay;            // Update total gross pay.


     InsFee = Insurance(InsPlan, Age); // Calculate insurance fee
     TotalIns += InsFee;                // Update total insurance
                                        //         fees.

     FIT = FedTax(GrossPay);            // Calculate income tax amt.
     TotalFIT += FIT;                   // Update total income
                                        //   taxes.

     SSI = GrossPay * SSRate;           // Calculate SS tax.
     TotalSSI += SSI;                   // Update total SS taxes.

     NetPay = GrossPay - InsFee -
                          FIT - SSI;    // Calculate net pay.
     TotalNet += NetPay;                // Update total net pay.

     // Write results to output file:
     PrintLine(outPay, IdNum, GrossPay, InsFee, FIT, SSI, NetPay);

     // Read next line of input:
     inPay.ignore(200, '\n');           // Skip to next line.
     inPay >> IdNum >> Age >> GrossPay >> InsPlan;
  }                                     // End of while loop.
```



```
// Print out averages (if appropriate):
  PrintAvg(outPay, TotalGross, TotalIns, TotalFIT, TotalSSI,
           TotalNet, NumEmployees);

// Close files.
  inPay.close();
  outPay.close();

  return NumEmployees;
}                            // end of main()
```



```
////////////////////////////////////////////////////////////////
//  Function to calculate the proper insurance fee for an employee.
//  Parameters:
//      InsPlan     character 'B' or 'D', indicating plan selected
//      Age         employee's age
//
//  The function doesn't need to modify the value of either of these
//  parameters, so they are passed in by value (no ampersand).
//
//  Return value:
//      InsFee      insurance fee charged to employee
//
//  Pre:  Plan and Age have been properly initialized
//  Post: corresponding insurance fee has been returned
//
double Insurance(char Plan, int Age) {
  const double HiBasic   = 150.00;   // constants for insurance fees
  const double MidBasic  = 100.00;
  const double LoBasic   =  70.00;
  const double HiDeluxe  = 250.00;
  const double MidDeluxe = 180.00;
  const double LoDeluxe  = 140.00;
  const char  Basic  = 'B';          // constants for plan types
  const char  Deluxe = 'D';
  double Fee = 0.0;                  // insurance fee charged
```



```
   switch (Plan) {                    // Calculate insurance fee
   case Basic:{ if (Age <= 35)     //     for Basic Plan
                       Fee = LoBasic;
                else if (Age <= 65)
                   Fee = MidBasic;
                else
                   Fee = HiBasic;
                break;
              }
   case Deluxe:{ if (Age <= 35)         //     for Deluxe Plan
                    Fee = LoDeluxe;
                 else if (Age <= 65)
                    Fee = MidDeluxe;
                 else
                    Fee = HiDeluxe;
                 break;
               }
   }
  return Fee;
}
```



```
//////////////////////////////////////////////////////////////////
//  Function to calculate the proper income tax for an employee.
//  Parameter:
//      GrossPay    employee's gross pay amount
//
//  Return value:
//      FIT         income tax withheld for employee
//
//  Pre:  Gross has been properly initialized
//  Post: corresponding income tax amount has been returned
//
double FedTax(double Gross) {
  const double HiTax  = 0.33;          // tax rates
  const double MidTax = 0.28;
  const double LoTax  = 0.15;

  const double HiIncome  = 3000.00;    // income levels
  const double MidIncome = 1000.00;

  double FIT = 0.0;                    // income tax withheld

  if (Gross <= MidIncome) {            // Determine FIT amount.
     FIT = LoTax * Gross;                 // low income
  }
  else if (Gross <= HiIncome) {
     FIT = LoTax * MidIncome              // middle income
        + MidTax * (Gross - MidIncome);
  }
  else {
     FIT = LoTax * MidIncome              // high income
        + MidTax * (HiIncome - MidIncome)
        + HiTax * (Gross - HiIncome);
  }
  return FIT;
}
```



```
////////////////////////////////////////////////////////////////
//  Function to print header for output file.
//  Parameter:
//    outPay      input file stream variable
//
//  The function does need to modify this stream (by inserting to
//  it), so the parameter must be passed by reference (with an
//  ampersand).
//
//  The function doesn't need to return a value, so it's void.
//
//  Pre:  output file has been opened
//  Post: file header has been written
//
void PrintHeader(ofstream& outPay) {
  outPay << "Programmer:  Bill McQuain" << endl;
  outPay << "StarFleet Payroll" << endl << endl;
  outPay << "   ID    Gross Pay     FIT       SSI       Ins"
         << "       Net Pay";
  outPay << endl;
  outPay << "============================================="
         << "==============";
  outPay << endl;
}
```



```
//////////////////////////////////////////////////////////////////
//  Function to print a line of employee data to output file.
//  Parameter:
//      outPay      input file stream variable
//      IdNum        id number of employee
//      GrossPay    gross pay for employee
//      InsFee      insurance fee for employee
//      FIT         income tax for employee
//      SSI         social security tax for employee
//      NetPay      net pay for employee
//
//  The function does need to modify this stream (by inserting to
//  it), so the parameter must be passed by reference (with an
//  ampersand).
//
//  Pre:  output file has been opened; other parameters have been
//        properly initialized
//  Post: formatted line of output has been written
//
void PrintLine(ofstream& outPay, int IdNum, double Gross,
               double Fee, double FIT, double SSI, double Net) {
  outPay << setw( 6) << IdNum;
  outPay << setw(12) << setprecision(2) << Gross;
  outPay << setw(10) << setprecision(2) << FIT;
  outPay << setw(10) << setprecision(2) << SSI;
  outPay << setw(10) << setprecision(2) << Fee;
  outPay << setw(12) << setprecision(2) << Net;
  outPay << endl;
}
```



```
//////////////////////////////////////////////////////////////////
//  Function to print final averages to input file.
//  Parameter:
//    outPay         input file stream variable
//      TotalGross     total gross pay (for all employees)
//      TotalIns       total insurance fees
//      TotalFIT       total income tax
//      TotalSSI       total social security tax
//      TotalNet       total net pay
//      NumEmployees   total number of employees
//
//  The function does need to modify this stream (by inserting to
//  it), so the parameter must be passed by reference (with an
//  ampersand).
//
//  Pre:  output file has been opened; other parameters have been
//        properly initialized
//  Post: formatted averages have been printed, if number of
//        employees is positive
//
void PrintAvg(ofstream& outPay, double TotalGross, double TotalIns,
              double TotalFIT, double TotalSSI, double TotalNet,
              int NumEmployees) {
   outPay << "======================================"
          << "====================="
          << endl;
   if (NumEmployees > 0) {
      outPay << "  Avg:" << setprecision(2);
      outPay << setw(12) << TotalGross/NumEmployees;
      outPay << setw(10) << TotalFIT/NumEmployees;
      outPay << setw(10) << TotalSSI/NumEmployees;
      outPay << setw(10) << TotalIns/NumEmployees;
      outPay << setw(12) << TotalNet/NumEmployees << endl;
   }
}
```



```
////////////////////////////////////////////////////////////
//  Title:         Payroll Program with Arrays
//  Programmer:    Bill McQuain
//  ID Number:     999-99-9999
//  Compiler:      Microsoft Visual C++ ver 6.0
//  Platform:      Pentium II 400, Windows NT
//  Last modified: January 2, 1999
//
//  Purpose:
//
//    Same as the original version, Pay99.cpp.
//
//  Implementation Issues:
//
//    The principal alteration is that this program uses a
//    collection of parallel arrays to store all the data.
//    An assumption is made as to the maximum number of
//    employees for whom data will be processed; for ease
//    of modification, that value is set as a file-scoped
//    constant.
//
//    By "parallel" I mean that array elements with the
//    same index will hold information about the same
//    employee.
//
//    Other changes (from FuncPay99.cpp):
//
//    - The overall structure of main() is now divided
//      into logical sections:
//          - declarations
//          - function call to read data
//          - function call to calculate taxes, insurance,
//            net pay
//          - function call to print the table, including
//            header and averages
//
```



```
//    - The input file is needed only by the function
//      ReadData() and the output file is needed only by
//      the function PrintTable() and the functions
//      PrintHeader() and CalcAvgs(), which are called by
//      PrintTable().  Therefore, the input and output file
//      stream variables are made local to ReadData() and
//      PrintTable() respectively.
//
#include <fstream.h>
#include <iomanip.h>

const int MaxEmp = 20;     // maximum number of employees

//  Function prototypes.  It's not necessary to include names for
//  the parameters here, but I recommend it because it helps keep
//  things straight.

double FedTax(double EmpGrossPay);
double Insurance(char EmpInsPlan, int EmpAge);
void   PrintHeader(ofstream& outPay);
void   ReadData(int IdNum[], double GrossPay[],
                int Age[], char InsPlan[], int& NumEmp);
void   CalcData(const double GrossPay[], const int Age[],
                const char InsPlan[], double InsFee[], double FIT[],
                double SSI[], double NetPay[], int NumEmp);
void   PrintTable(const int IdNum[], const double GrossPay[],
                        const double InsFee[], const double FIT[],
                  const double SSI[],
                  const double NetPay[],int NumEmp);
void   CalcAvgs(ofstream& outPay, const double GrossPay[],
                const double InsFee[],
                const double FIT[], const double SSI[],
                const double NetPay[], int NumEmp);
```



```
////////////////////////////////////////////////////////////
//  Beginning of main function:
//
int main() {

//  Local variables for main():

  int    IdNum[MaxEmp],       // employee ID number
         Age[MaxEmp],         // employee age
         NumEmp;              // number of employees
  double GrossPay[MaxEmp],    // employee gross pay
         NetPay[MaxEmp],      // employee net pay
         SSI[MaxEmp],         // employee SS tax
         FIT[MaxEmp],         // employee income tax
         InsFee[MaxEmp];      // employee insurance fee
  char   InsPlan[MaxEmp];     // employee insurance plan

// Read employee data from input file:

  ReadData(IdNum, GrossPay, Age, InsPlan, NumEmp);

// Calculate employee withholding data and net pay:

  CalcData(GrossPay, Age, InsPlan, InsFee, FIT, SSI,
           NetPay, NumEmp);

// Print the table of employee information:

  PrintTable(IdNum, GrossPay, InsFee, FIT, SSI, NetPay,
             NumEmp);

  return NumEmp;
}                            // end of main()
```



```
//////////////////////////////////////////////////////////////
//  The function ReadData reads employee data from the input
//  file and stores it in the array parameters.  ReadData will
//  read until the end of the input file or until the arrays
//  are filled (determined by the global constant MaxEmp).
//
//  Output parameters (passed by reference):
//
//    IdNum[]        list of employee ID numbers
//    GrossPay[]     list of gross pay amounts
//    Age[]          list of employee ages
//    InsPlan[]      list of insurance plan codes
//    NumEmp         number of employees for whom data was
//                   stored
//
//  Remember:  An array parameter is passed by reference
//             by default, so no ampersands (&) are needed
//             for them.
//
//  Also note: We don't need to specify the dimension of an
//             array parameter in the function definition
//             (or in the prototype).  But you do need to
//             indicate that it's an array; that's the reason
//             for the []s.
//
//  ReadData must be able to handle the situation where
//  there are more lines of input data than will fit into
//  the arrays.  This is done by using two mechanisms.  First,
//  the loop control for the while includes both a check for a
//  read failure and a check to be sure that we're not about
//  to try to store values past the end of the arrays.  Second,
//  we read the input values into local variables first, then
//  transfer them to the arrays only if the array index is
//  valid (MaxEmp - 1 or less).
```



```
//  One point that may be confusing.  The local variable Index
//  is used to keep track of which array locations are to be
//  stored into next. When we exit the while loop, Index will
//  point to the first unfilled location in the arrays.
//  However, Index will equal the number of employees whose
//  records were stored into the arrays.  Remember that in C++
//  we index arrays starting at 0 instead of 1.
//
//  Pre:  none
//  Post: up to MaxEmp employee data lines have been read from
//        the input file; data stored in IdNum[], GrossPay[],
//        Age[] and InsPlan[]; number of employee data lines
//        read stored in NumEmp.
//
void ReadData(int IdNum[], double GrossPay[], int Age[],
              char InsPlan[], int& NumEmp) {

  int    Index = 0; // next array location to fill

  int    empnum,    // temp storage for employee id number
         empage;    //      employee age,
  double gross;     //      gross pay, and
  char   plan;      //      insurance plan code
  ifstream inPay;   // input file stream

  inPay.open("workers.dat");       // open input file

  // Read first line of data from input file:

  inPay >> empnum >> empage >> gross >> plan;
```



```
  // Process and read next line until there's no more data or
  // we've filled up the arrays:

  while (inPay && Index < MaxEmp) {

     inPay.ignore(200, '\n');    // Skip to next line.

     IdNum[Index]    = empnum;   // Store the last input line
     Age[Index]      = empage;   //    in the relevant arrays
     GrossPay[Index] = gross;
     InsPlan[Index]  = plan;

     inPay >> empnum >> empage >> gross >> plan;

     Index++;        // Update index for next store
  }

  NumEmp = Index;    // Remember: we never stored anything
                     //  after the last increment of Index

  inPay.close();     // Close the input file.
}
```



```
//////////////////////////////////////////////////////////////
//  CalcData is used to calculate the net pay amounts for all
//  employees.
//
//  Input Parameters (passed by value):
//
//    GrossPay[]     list of gross pay amounts
//    Age[]          list of employee ages
//    InsPlan[]      list of insurance plan codes
//    NumEmp         number of employees to be processed
//
//  Note:  to pass array parameters by value we specify their
//         type using "const" --- both in the definition and
//         in the prototype.  This is really "pass by constant
//         reference" as we'll discuss in class.  For
//         practical purposes, the effect is the same as pass
//         by value.
//
//  Output Parameters (passed by reference):
//
//    InsFee[]       list of insurance fees
//    FIT[]          list of federal income taxes
//    SSI[]          list of social security taxes
//    NetPay[]       list of net pay amounts
//
//  Pre:  GrossPay[], Age[], InsPlay[], InsFee[] initialized;
//        NumEmp equals number of employees
//  Post: FIT[], SSI[] and NetPay[] properly calculated
//
void CalcData(const double GrossPay[], const int Age[],
              const char InsPlan[], double InsFee[], double FIT[],
              double SSI[], double NetPay[], int NumEmp){

   const double SSRate = 0.0683;           // SSI tax rate
```



```
    ////////////////////////////////////////////////////////////
   //  Calculate the insurance fee, income tax, social security
   //  tax and net pay for each employee, storing the results in
   //  the relevant arrays:
   //
   for (int count = 0; count < NumEmp; count++) {

      InsFee[count] = Insurance(InsPlan[count], Age[count]);
      FIT[count]    = FedTax(GrossPay[count]);
      SSI[count]    = SSRate * GrossPay[count];
      NetPay[count] = GrossPay[count] - FIT[count] - SSI[count] -
                      InsFee[count];
   }
}
```



```
//////////////////////////////////////////////////////////////////
//  Function to calculate the proper insurance fee for an employee.
//  Parameters:
//      InsPlan     character 'B' or 'D', indicating plan selected
//      Age         employee's age
//
//  The function doesn't need to modify the value of either of these
//  parameters, so they are passed in by value (no ampersand).
//
//  Return value:
//      InsFee      insurance fee charged to employee
//
//  Pre:  Plan and Age have been properly initialized
//  Post: corresponding insurance fee has been returned
//
double Insurance(char Plan, int Age) {

  const double HiBasic   = 150.00;   // constants for insurance fees
  const double MidBasic  = 100.00;
  const double LoBasic   =  70.00;
  const double HiDeluxe  = 250.00;
  const double MidDeluxe = 180.00;
  const double LoDeluxe  = 140.00;

  const char  Basic  = 'B';          // constants for plan types
  const char  Deluxe = 'D';

  double Fee = 0.0;                  // insurance fee charged
```



```
 switch (Plan) {                    // Calculate insurance fee
  case Basic:{ if (Age <= 35)         //     for Basic Plan
                  Fee = LoBasic;
               else if (Age <= 65)
                  Fee = MidBasic;
               else
                  Fee = HiBasic;
               break;
             }
  case Deluxe:{ if (Age <= 35)        //     for Deluxe Plan
                   Fee = LoDeluxe;
                else if (Age <= 65)
                   Fee = MidDeluxe;
                else
                   Fee = HiDeluxe;
                break;
              }
  }

  return Fee;
}
```



```
////////////////////////////////////////////////////////////////
//  Function to calculate the proper income tax for an employee.
//  Parameter:
//      GrossPay    employee's gross pay amount
//  Return value:
//      FIT         income tax withheld for employee
//  Pre:  Gross has been properly initialized
//  Post: corresponding income tax amount has been returned
//
double FedTax(double Gross) {

  const double HiTax  = 0.33;          // tax rates
  const double MidTax = 0.28;
  const double LoTax  = 0.15;

  const double HiIncome  = 3000.00;    // income levels
  const double MidIncome = 1000.00;

  double FIT = 0.0;                    // income tax withheld

  if (Gross <= MidIncome) {            // Determine FIT amount.
     FIT = LoTax * Gross;                   // low income
  }
  else if (Gross <= HiIncome) {
     FIT = LoTax * MidIncome                // middle income
        + MidTax * (Gross - MidIncome);
  }
  else {
     FIT = LoTax * MidIncome                // high income
        + MidTax * (HiIncome - MidIncome)
        + HiTax * (Gross - HiIncome);
  }

  return FIT;
}
```



```
//////////////////////////////////////////////////////////////////
//  Function to print header for output file.
//  Parameter:
//    outPay      input file stream variable
//
//  The function does need to modify this stream (by inserting to
//  it), so the parameter must be passed by reference (with an
//  ampersand).
//
//  The function doesn't need to return a value, so it's void.
//
//  Pre:  output file has been opened
//  Post: file header has been written
//
void PrintHeader(ofstream& outPay) {

  outPay << "Programmer:  Bill McQuain" << endl;
  outPay << "StarFleet Payroll" << endl << endl;
  outPay << "   ID    Gross Pay     FIT       SSI       Ins"
         << "       Net Pay";
  outPay << endl;
  outPay << "=============================================="
         << "==============";
  outPay << endl;

}
```



```
//////////////////////////////////////////////////////////////////
//  PrintTable is used to print the table of employee data to output
//  file.
//
//  Input Parameters (passed by value):
//
//      IdNum        id numbers of employees
//      GrossPay    gross pay for employees
//      InsFee      insurance fees for employees
//      FIT         income taxes for employees
//      SSI         social security taxes for employees
//      NetPay      net pay for employees
//      NumEmp      number of employees to print data for
//
//  Pre:  arrays hold proper data
//  Post: table of payroll data has beeen printed to output file
//
void  PrintTable(const int IdNum[], const double GrossPay[],
                 const double InsFee[], const double FIT[],
                 const double SSI[], const double NetPay[],
                 int NumEmp) {

  ofstream outPay;               // output file stream

  outPay.open("payola.dat");  // open the output file
                              // Set up for floating point output:
  outPay.setf(ios::fixed, ios::floatfield);
  outPay.setf(ios::showpoint);
```



```
  // Print the header for the table:
  //
  PrintHeader(outPay);

  // Print the table, line by line:
  //
  for (int count = 0; count < NumEmp; count++) {

     outPay << setw( 6) << IdNum[count];
     outPay << setw(12) << setprecision(2) << GrossPay[count];
     outPay << setw(10) << setprecision(2) << FIT[count];
     outPay << setw(10) << setprecision(2) << SSI[count];
     outPay << setw(10) << setprecision(2) << InsFee[count];
     outPay << setw(12) << setprecision(2) << NetPay[count];
     outPay << endl;
  }

  // Calculate and print averages:
  //
  CalcAvgs(outPay, GrossPay, InsFee, FIT, SSI, NetPay, NumEmp);
}
```



```
//////////////////////////////////////////////////////////////////
//  CalcAvgs is used to print final averages to input file.
//
//  Output Parameter (passed by reference):
//
//    outPay         output file stream variable
//
//  Input Parameters (passed by value):
//
//      TotalGross     total gross pay (for all employees)
//      TotalIns       total insurance fees
//      TotalFIT       total income tax
//      TotalSSI       total social security tax
//      TotalNet       total net pay
//      NumEmployees   total number of employees
//
//  Pre:  output file is open; arrays hold proper data
//  Post: if number of employees is positive, averages
//        have been printed; otherwise zeros are printed
//
void CalcAvgs(ofstream& outPay, const double GrossPay[],
              const double InsFee[],
              const double FIT[], const double SSI[],
              const double NetPay[], int NumEmp) {

 double TotalGross = 0.0,     // total of all gross pay
        TotalNet   = 0.0,     // total of all net pay
        TotalIns   = 0.0,     // total of all insurance fees
        TotalSSI   = 0.0,     // total of all SS tax
        TotalFIT   = 0.0;     // total of all income tax

  // Print out footer for table:

  outPay << "=============================================="
         << "================" << endl;
```



```
  // If NumEmp > 0 then calculate totals and print out averages;
  // otherwise print zeros for the averages.
  //
  if (NumEmp) {

     for (int count = 0; count < NumEmp; count++) {
        TotalGross += GrossPay[count];
        TotalIns   += InsFee[count];
        TotalFIT   += FIT[count];
        TotalSSI   += SSI[count];
        TotalNet   += NetPay[count];
     }
     outPay << "  Avg:" << setprecision(2);
     outPay << setw(12) << TotalGross/NumEmp;
     outPay << setw(10) << TotalFIT/NumEmp;
     outPay << setw(10) << TotalSSI/NumEmp;
     outPay << setw(10) << TotalIns/NumEmp;
     outPay << setw(12) << TotalNet/NumEmp;
     outPay << endl;
  }
  else {
     outPay << "  Avg:" << setprecision(2);
     outPay << setw(12) << 0.0;
     outPay << setw(10) << 0.0;
     outPay << setw(10) << 0.0;
     outPay << setw(10) << 0.0;
     outPay << setw(12) << 0.0;
     outPay << endl;
  }
}
```


Adding Sorting

SortPay.cpp incorporates an additional function to ArrayPay.cpp, producing an output file that is sorted by net pay amounts. Since the only alterations are the inclusion of a prototype, a function definition, and a call to that function from main( ), only the new function definition is provided here.

Sample input and output files:

```
3175 45 7597.43 B McCoy, Leonard
5328 47 3000.00 D Checkov, Pavel
3236 35 3897.89 B Sulu, Hikaru
2334 49 2089.10 D Riley, Kevin
5315 73 9336.00 B Kirk, James
7641 58 5409.12 D Uhura
7077 23 1001.00 B Rand, Janice
4098 32 8594.98 D Spock
3424 39 2999.99 B Solo, Han
4297 52 1000.00 B Chewbacca
1716 53 2011.50 D Skywalker, Luke
5956 67 5245.42 D Scott, Montgomery
```

```
Bill McQuain
Macro$oft Corporation Payroll

   ID    Gross Pay     FIT       SSI       Ins       Net Pay
=============================================================
  4297     1000.00    150.00     68.30    100.00      681.70
  7077     1001.00    150.28     68.37     70.00      712.35
  1716     2011.50    433.22    137.39    180.00     1260.89
  2334     2089.10    454.95    142.69    180.00     1311.47
  5328     3000.00    710.00    204.90    180.00     1905.10
  3424     2999.99    710.00    204.90    100.00     1985.09
  3236     3897.89   1006.30    266.23     70.00     2555.36
  5956     5245.42   1450.99    358.26    250.00     3186.17
  7641     5409.12   1505.01    369.44    180.00     3354.67
  3175     7597.43   2227.15    518.90    100.00     4751.37
  4098     8594.98   2556.34    587.04    140.00     5311.60
  5315     9336.00   2800.88    637.65    150.00     5747.47
=============================================================
  Avg:     4348.54   1179.59    297.00    141.67     2730.27
```


Adding Sorting

```
//////////////////////////////////////////////////////////////////
//  SortData sorts the array of NetPay amounts into increasing
//  order, rearranging the data in the other arrays to keep
//  everything consistent.  The selection sort algorithm is used.
//
//  Input Parameters (passed by value):
//
//    IdNum       id numbers of employees
//    GrossPay    gross pay for employees
//    InsFee      insurance fees for employees
//    FIT         income taxes for employees
//    SSI         social security taxes for employees
//    NetPay      net pay for employees
//    Age         ages of employees
//    InsPlan     insurance plan codes for employees
//    NumEmp      number of employees
//
//  Pre:  all parameters are properly initialized; NumEmp equals
//        number of employees being processed
//  Post: NetPay[] is sorted in ascending order; all other arrays
//        are maintained in proper relative order
//

void  SortData(int IdNum[], double GrossPay[], double InsFee[],
               double FIT[], double SSI[], double NetPay[],
               int Age[], char InsPlan[], int NumEmp) {

   int Begin,        // beginning index for current pass
       Check,        // index currently being checked
       SmallSoFar;   // index of largest entry found so far

   int    TmpInt;    // temporary storage used for swapping values
   double TmpFloat;
   char   TmpChar;
```


Adding Sorting

```
// The outer loop runs us down the list NumEmp - 1 times:
//
for (Begin = 0; Begin < NumEmp - 1; Begin++) {

   SmallSoFar = Begin;     // smallest so far is the first

   //  The inner loop runs down the "tail" of the list,
   //  determining where the smallest value is located.
   //
   //  The value found will belong at the head of the "tail".
   //
   for (Check = Begin; Check < NumEmp; Check++) {
      if (NetPay[Check] < NetPay[SmallSoFar])
         SmallSoFar = Check;
   }

   //  Swap the smallest net pay amount located by the inner
   //  loop to the right index:
   //
   TmpFloat           = NetPay[SmallSoFar];
   NetPay[SmallSoFar] = NetPay[Begin];
   NetPay[Begin]      = TmpFloat;

   //  ... and the corresponding (not the smallest) ID number:
   //
   TmpInt             = IdNum[SmallSoFar];
   IdNum[SmallSoFar] = IdNum[Begin];
   IdNum[Begin]      = TmpInt;

   //  ... and the corresponding gross pay amount:
   //
   TmpFloat              = GrossPay[SmallSoFar];
   GrossPay[SmallSoFar] = GrossPay[Begin];
   GrossPay[Begin]      = TmpFloat;
```



```
      //  ... and the corresponding insurance fee:
      //
      TmpFloat           = InsFee[SmallSoFar];
      InsFee[SmallSoFar] = InsFee[Begin];
      InsFee[Begin]      = TmpFloat;

      //  ... and the corresponding income tax amount:
      //
      TmpFloat        = FIT[SmallSoFar];
      FIT[SmallSoFar] = FIT[Begin];
      FIT[Begin]      = TmpFloat;

      //  ... and the corresponding social security amount:
      //
      TmpFloat        = SSI[SmallSoFar];
      SSI[SmallSoFar] = SSI[Begin];
      SSI[Begin]      = TmpFloat;

      //  ... and the corresponding age:
      //
      TmpInt          = Age[SmallSoFar];
      Age[SmallSoFar] = Age[Begin];
      Age[Begin]      = TmpInt;

      //  ... and finally, the corresponding insurance plan
      //      code:
      //
      TmpChar             = InsPlan[SmallSoFar];
      InsPlan[SmallSoFar] = InsPlan[Begin];
      InsPlan[Begin]      = TmpChar;

   }
}
```



```
//////////////////////////////////////////////////////////////
//  Title:         Simple Payroll with an array of Structs
//  Programmer:    Bill McQuain
//  ID Number:     999-99-9999
//  Compiler:      Microsoft Visual C++ ver 6.0
//  Platform:      Pentium II 400, Windows NT
//  Last modified: January 2, 1999
//
//  Purpose:
//
//    Same as the original version, Pay99.cpp.
//
//  Implementation Issues:
//
//  The only modifications from ArrayPay99.cpp are related to
//  the use of an array of struct variables to hold the
//  employee records, replacing the bouquet of parallel arrays
//  used before.  The primary points of interest are the vastly
//  simpler parameter lists that are needed now, and the
//  relatively trivial swap section of the sorting code.
//
#include <fstream.h>
#include <iomanip.h>

const int MaxEmp = 20;  // maximum number of employees

typedef struct {
   int    IdNum;        // employee's ID number
   int    Age;          // employee's age
   char   InsPlan;      // employee's insurance plan code
   double GrossPay;     // employee's gross pay
   double InsFee;       // employee's insurance fee
   double FIT;          // employee's federal income tax
   double SSI;          // employee's social security tax
   double NetPay;       // employee's net pay
} EmpRec;
```



```
//  Function prototypes:
void   ReadData(EmpRec Emp[], int& NumEmp);
void   CalcData(EmpRec Emp[], int NumEmp);
double FedTax(double EmpGrossPay);
double Insurance(char EmpInsPlan, int EmpAge);
void   SortData(EmpRec Emp[], int NumEmp);
void   PrintTable(const EmpRec Emp[],int NumEmp);
void   PrintHeader(ofstream& outPay);
void   CalcAvgs(ofstream& outPay, const EmpRec Emp[], int NumEmp);

////////////////////////////////////////////////////////////
//  Beginning of main function:
int main() {
//  Local variables for main():
  EmpRec Emp[MaxEmp];         // array of employee records
  int   NumEmp;               // number of employees

// Read employee data from input file:

  ReadData(Emp, NumEmp);

// Calculate employee withholding data and net pay:

  CalcData(Emp, NumEmp);

// Sort the data so net pay amounts are in increasing order:

  SortData(Emp, NumEmp);

// Print the table of employee information:

  PrintTable(Emp, NumEmp);

  return NumEmp;
}                              // end of main()
```



```
//////////////////////////////////////////////////////////////
//  The function ReadData reads employee data from the input
//  file and stores it in the array parameters.  ReadData will
//  read until the end of the input file or until the arrays
//  are filled (determined by the global constant MaxEmp).
//
//  Output parameters (passed by reference):
//
//    Emp[]     array of employee records
//    NumEmp    number of employees for whom data was stored
//
//  Pre:  none
//  Post: up to MaxEmp employee data lines have been read from
//        the input file; data stored in employee records in
//        Emp[]; number of employee data lines read stored in
//        NumEmp.
//
void ReadData(EmpRec Emp[], int& NumEmp) {

  int    Index = 0;   // next array location to fill

  int    empnum,      // temp storage for employee id number
         empage;      //      employee age,
  double gross;       //      gross pay, and
  char   plan;        //      insurance plan code

  ifstream inPay;                 // input file stream

  inPay.open("workers.dat");  // open input file

  // Read first line of data from input file:

  inPay >> empnum >> empage >> gross >> plan;
```



```
  // Process and read next line until there's no more data or
  // we've filled up the arrays:

  while (inPay && Index < MaxEmp) {

     inPay.ignore(200, '\n');       // Skip to next line.

     Emp[Index].IdNum    = empnum;  // Store the last input line
     Emp[Index].Age      = empage;  //     in an employee record
     Emp[Index].GrossPay = gross;
     Emp[Index].InsPlan  = plan;

     inPay >> empnum >> empage >> gross >> plan;

     Index++;                       // Update index for next store
  }

  NumEmp = Index;   // Remember: we never stored anything after
                    //           the last increment of Index

  inPay.close();    // Close the input file.
}
```



```
//////////////////////////////////////////////////////////////
//  CalcData is used to calculate the net pay amounts for all
//  employees.
//
//  Input Parameters (passed by value):
//
//    NumEmp         number of employees to be processed
//
//  Input/Output Parameters (passed by reference):
//
//    Emp[]          array of employee records
//
//  Pre:  Emp[] stores NumEmp employee records, holding data
//        for gross pay, age, id number, and insurance plan
//  Post: Emp[] updated for insurance fees, tax amounts, and
//        net pay amounts
//
void CalcData(EmpRec Emp[], int NumEmp){

   const double SSRate = 0.0683;       // SSI tax rate

   //////////////////////////////////////////////////////////
   //  Calculate the insurance fee, income tax, social security
   //  tax and net pay for each employee, storing the results in
   //  the relevant arrays:
   //
   for (int count = 0; count < NumEmp; count++) {

      Emp[count].InsFee = Insurance(Emp[count].InsPlan,
                                    Emp[count].Age);
      Emp[count].FIT    = FedTax(Emp[count].GrossPay);
      Emp[count].SSI    = SSRate * Emp[count].GrossPay;
      Emp[count].NetPay = Emp[count].GrossPay - Emp[count].FIT -
                          Emp[count].SSI - Emp[count].InsFee;
   }
}
```



```
////////////////////////////////////////////////////////////
//  Insurance is used to calculate the proper insurance fee for
//  an employee.
//
//  Input Parameters (passed by value):
//
//    InsPlan     character 'B' or 'D', indicating plan selected
//    Age         employee's age
//
//  Return value:
//
//    InsFee          insurance fee charged to employee
//
float Insurance(char InsPlan, int Age) {

//  Body omitted --- identical to version in ArrayPay.cpp
//

////////////////////////////////////////////////////////////
//  FedTax is used to calculate the proper income tax for an
//  employee.
//
//  Input Parameter (passed by value):
//
//        GrossPay  employee's gross pay amount
//
//  Return value:
//
//        FIT       income tax withheld for employee
//
float FedTax(float GrossPay) {

//  Body omitted --- identical to version in ArrayPay.cpp
//
```



```
//////////////////////////////////////////////////////////////
//  SortData sorts the array of NetPay amounts into increasing
//  order, rearranging the data in the other arrays to keep
//  everything consistent.  The selection sort algorithm is used.
//
//  Input Parameters (passed by value):
//
//    NumEmp      number of employees
//
//  Input/Output parameters (passed by reference):
//
//    Emp[]       array of employee records
//
//  Pre:  all parameters are properly initialized; NumEmp equals
//        number of employees being processed
//  Post: Emp[] is sorted in ascending order by net pay
//
void  SortData(EmpRec Emp[], int NumEmp) {

   int Begin,        // beginning index for current pass
       Check,        // index currently being checked
       SmallSoFar;   // index of largest entry found so far

   EmpRec TmpEmp;    // temporary storage used for swapping
```



```
   // The outer loop runs us down the list NumEmp - 1 times:
   //
   for (Begin = 0; Begin < NumEmp - 1; Begin++) {

      SmallSoFar = Begin;     // smallest so far is the first

      //  The inner loop runs down the "tail" of the list,
      //  determining where the smallest value is located.
      //
      //  The value found will belong at the head of the "tail".
      //
      for (Check = Begin; Check < NumEmp; Check++) {
         if (Emp[Check].NetPay < Emp[SmallSoFar].NetPay)
            SmallSoFar = Check;
      }

      //  Swap the employee record with smallest net pay amount,
      //  located by the inner loop, to the right index:
      //
      TmpEmp          = Emp[SmallSoFar];
      Emp[SmallSoFar] = Emp[Begin];
      Emp[Begin]      = TmpEmp;
   }
}
```



```
//////////////////////////////////////////////////////////////
//  PrintTable is used to print the table of employee data to
//  output file.
//
//  Input Parameters (passed by value):
//
//    Emp[]       array of employee records
//    NumEmp      number of employees to print data for
//
//  Pre:  Emp[] holds NumEmp employee records
//  Post: table of payroll data has been printed
//
void  PrintTable(const EmpRec Emp[], int NumEmp) {

  ofstream outPay;            // output file stream
  outPay.open("payola.dat");  // open the output file
                              // Set up for decimal output:
  outPay.setf(ios::fixed, ios::floatfield);
  outPay.setf(ios::showpoint);

  // Print the header for the table:
  PrintHeader(outPay);

  // Print the table, line by line:
  for (int count = 0; count < NumEmp; count++) {
     outPay << setw( 6) << Emp[count].IdNum;
     outPay << setw(12) << setprecision(2) << Emp[count].GrossPay;
     outPay << setw(10) << setprecision(2) << Emp[count].FIT;
     outPay << setw(10) << setprecision(2) << Emp[count].SSI;
     outPay << setw(10) << setprecision(2) << Emp[count].InsFee;
     outPay << setw(12) << setprecision(2) << Emp[count].NetPay;
     outPay << endl;
  }
  // Calculate and print averages:
  CalcAvgs(outPay, Emp, NumEmp);
}
```



```
////////////////////////////////////////////////////////////////
//  PrintHeader is used to print header for output file.
//
//  Output Parameter (passed by reference):
//
//        outPay         output file stream variable
//
void PrintHeader(ofstream& outPay) {

     //  no changes from ArrayPay.cpp
}

////////////////////////////////////////////////////////////////
//  CalcAvgs is used to print final averages to input file.
//
//  Output Parameter (passed by reference):
//
//    outPay         output file stream variable
//
//  Input Parameters (passed by value):
//
//    Emp[]          array of employee records
//        NumEmployees   total number of employees
//
//  Pre:  output file is open; Emp[] holds NumEmp records
//  Post: if number of employees is positive, averages
//        have been printed; otherwise zeros are printed
//
void CalcAvgs(ofstream& outPay, const EmpRec Emp[], int NumEmp) {

 double TotalGross = 0.0,     // total of all gross pay
        TotalNet   = 0.0,     // total of all net pay
        TotalIns   = 0.0,     // total of all insurance fees
        TotalSSI   = 0.0,     // total of all SS tax
        TotalFIT   = 0.0;     // total of all income tax
```



```
  // Print out footer for table:
  outPay << "=============================================="
         << "==============" << endl;

  // If NumEmp > 0 then calculate totals and print out averages;
  // otherwise print zeros for the averages.
  //
  if (NumEmp) {

     for (int count = 0; count < NumEmp; count++) {
        TotalGross += Emp[count].GrossPay;
        TotalIns   += Emp[count].InsFee;
        TotalFIT   += Emp[count].FIT;
        TotalSSI   += Emp[count].SSI;
        TotalNet   += Emp[count].NetPay;
     }
     outPay << "  Avg:" << setprecision(2);
     outPay << setw(12) << TotalGross/NumEmp;
     outPay << setw(10) << TotalFIT/NumEmp;
     outPay << setw(10) << TotalSSI/NumEmp;
     outPay << setw(10) << TotalIns/NumEmp;
     outPay << setw(12) << TotalNet/NumEmp;
     outPay << endl;
  }
  else {
     outPay << "  Avg:" << setprecision(2);
     outPay << setw(12) << 0.0;
     outPay << setw(10) << 0.0;
     outPay << setw(10) << 0.0;
     outPay << setw(10) << 0.0;
     outPay << setw(12) << 0.0;
     outPay << endl;
  }
}
```


NamesPayA.cpp incorporates code to read, store and print the employee names already included in the input file. In addition, a second input file indicates which employees will receive a bonus (a 15% increase in gross pay). A sample second input file and output are provided below.

The employee names in the bonus file are followed immediately by a tab.

Sample (second) input and output files:

```
Employee              Bonus?
Checkov, Pavel          Y
Chewbacca               N
Kirk, James             N
McCoy, Leonard          Y
Rand, Janice            Y
Riley, Kevin            N
Scott, Montgomery       Y
Skywalker, Luke         N
Solo, Han               N
Spock                   N
Sulu, Hikaru            N
Uhura                   Y
```

```
Programmer:  Bill McQuain
StarFleet Payroll


Name                IdNum   Gross Pay      FIT       SSI       Ins       Net Pay
=============================================================================
Chewbacca            4297     1000.00    150.00     68.30    100.00       681.70
Rand, Janice         7077     1151.15    192.32     78.62     70.00       810.20
Skywalker, Luke      1716     2011.50    433.22    137.39    180.00      1260.89
Riley, Kevin         2334     2089.10    454.95    142.69    180.00      1311.47
Solo, Han            3424     2999.99    710.00    204.90    100.00      1985.09
Checkov, Pavel       5328     3450.00    858.50    235.63    180.00      2175.86
Sulu, Hikaru         3236     3897.89   1006.30    266.23     70.00      2555.36
Uhura                7641     5409.12   1505.01    369.44    180.00      3354.67
Scott, Montgomery    5956     6032.23   1710.64    412.00    250.00      3659.59
Spock                4098     8594.98   2556.34    587.04    140.00      5311.60
McCoy, Leonard       3175     8737.04   2603.22    596.74    100.00      5437.08
Kirk, James          5315     9336.00   2800.88    637.65    150.00      5747.47
=============================================================================
Average                       4559.08   1248.45    311.39    141.67      2857.58
```



```
////////////////////////////////////////////////////////////
//  Title:         Payroll Program with Employee Names
//  Programmer:    Bill McQuain
//  ID Number:     000-00-0000
//  Compiler:      Microsoft Visual C++ ver 6.0
//  Platform:      Pentium II 400, Windows NT
//  Last modified: January 5, 1999
//
//  Purpose:
//
//    Same as the simple payroll program, Pay99.cpp; with two
//    extensions.  The employee names in the input file are no
//    longer ignored, and a second input file containing
//    employee bonus information is also processed.
//
//  Implementation Issues:
//
//    The only modifications from StructPay.cpp are related
//    to the inclusion of employee names and bonus flags.  To
//    accommodate that, a second array of structs is used to
//    hold bonus data, and a new function EarnedBonus() is used
//    to compare each employee's name to those in the bonus list
//    until a match is found, after which the bonus flag for
//    that employee is checked.
//
//    Character data is handled in C-style, using char arrays
//    and the associated library functions from string.h.
//
#include <fstream.h>
#include <iomanip.h>
#include <string.h>        // to use strcmp() and strcpy()

const int MaxEmp  = 20;    // maximum number of employees
const int NameLen = 35;    // maximum employee name length
```



```
typedef struct {
   int    IdNum;           // employee's ID number
   char   Name[NameLen];   // employee's name
   int    Age;             // employee's age
   char   InsPlan;         // employee's insurance plan code
   double GrossPay;        // employee's gross pay
   double InsFee;          // employee's insurance fee
   double FIT;             // employee's federal income tax
   double SSI;             // employee's social security tax
   double NetPay;          // employee's net pay
} EmpRec;

typedef struct {
   char Name[NameLen];    // employee name
   bool GetsBonus;        // true if yes, false if no
} BonusRec;

//  Function prototypes:

void   InitRecords(EmpRec Emp[]);
void   ReadPayData(EmpRec Emp[], int& NumEmp);
void   ReadBonusList(BonusRec Bonus[]);
void   CalcData(EmpRec Emp[], BonusRec Bonus[], int NumEmp);
double FedTax(double EmpGrossPay);
double Insurance(char EmpInsPlan, int EmpAge);
bool   EarnedBonus(const char Name[], const BonusRec Bonus[],
                   int NumEmp);
void   SortData(EmpRec Emp[], int NumEmp);
void   PrintTable(const EmpRec Emp[], int NumEmp);
void   PrintHeader(ofstream& outPay);
void   CalcAvgs(ofstream& outPay, const EmpRec Emp[],
                int NumEmp);
void   SkipWhiteSpace(ifstream& ifile);
```



```
////////////////////////////////////////////////////////////
//  Beginning of main function:
//
int main() {

//  Local variables for main():

  EmpRec   Emp[MaxEmp];       // array of employee records
  BonusRec Bonus[MaxEmp];     // array of bonus records
  int       NumEmp;               // number of employees

// Initialize employee records:

  InitRecords(Emp);

// Read employee pay data and bonus list from input files:

  ReadPayData(Emp, NumEmp);
  ReadBonusList(Bonus);

// Calculate employee withholding data and net pay:

  CalcData(Emp, Bonus, NumEmp);

// Sort the data so net pay amounts are in increasing order:

  SortData(Emp, NumEmp);

// Print the table of employee information:

  PrintTable(Emp, NumEmp);

  return NumEmp;
}
```



```
//////////////////////////////////////////////////////////////
//  The function ReadPayData reads employee data from the input
//  file and stores it in the array parameters.  ReadData will
//  read until the end of the input file or until the arrays
//  are filled (determined by the global constant MaxEmp).
//
//  Output parameters (passed by reference):
//
//    Emp[]       array of employee records
//    NumEmp      number of employees for whom data was stored
//
//
void ReadPayData(EmpRec Emp[], int& NumEmp) {

  int    Index = 0;   // next array location to fill

  int    empnum,           // temp storage for employee id number
         empage;           //      employee age,
  double gross;            //      gross pay,
  char   plan;             //      insurance plan code, and
  char   empname[NameLen]; //      name

  ifstream inPay;             // input file stream

  inPay.open("workers.dat");  // open input file

  // Read first line of data from input file:

  inPay >> empnum >> empage >> gross >> plan;

  SkipWhiteSpace(inPay);       // skip whitespace after plan code
  inPay.get(empname, NameLen); // read employee name
```



```
  // Process and read next line until there's no more data or
  // we've filled up the arrays:

  while (inPay && Index < MaxEmp) {

     Emp[Index].IdNum    = empnum;   // Store the last input line
     Emp[Index].Age      = empage;   //       in an employee record
     Emp[Index].GrossPay = gross;
     Emp[Index].InsPlan  = plan;
     strcpy(Emp[Index].Name, empname);

     inPay.ignore(200, '\n');          // Skip to next line.

     inPay >> empnum >> empage >> gross >> plan;

     SkipWhiteSpace(inPay);       // skip whitespace after plan code
     inPay.get(empname, NameLen); // read employee name

     Index++;                     // Update index for next store
  }

  NumEmp = Index;       // Remember: we never stored anything after
                        //           the last increment of Index

  inPay.close();        // Close the input file.
}
```



```
//////////////////////////////////////////////////////////////
//  ReadBonusList reads list of employee names and flags
//  indicating whether each employee is to receive a bonus
//  during this pay period.
//
//  Output parameter:
//    Bonus[]     array of bonus records
//
void  ReadBonusList(BonusRec Bonus[]) {

   int  count = 0;
   char tmpName[NameLen];
   char Flag;
   ifstream inBonus;
   inBonus.open("bonus.dat");

   inBonus.ignore(80, '\n');            // skip header line

   inBonus.get(tmpName, NameLen, '\t');
   inBonus >> Flag;
   SkipWhiteSpace(inBonus);
   count++;

   while (inBonus && count < MaxEmp) {

      strcpy(Bonus[count].Name, tmpName);
      Bonus[count].GetsBonus = (Flag == 'Y');

      SkipWhiteSpace(inBonus);
      inBonus.get(tmpName, NameLen, '\t');
      inBonus >> Flag;
      count++;
   }
   inBonus.close();
}
```



```
//////////////////////////////////////////////////////////////
//  CalcData is used to calculate the net pay amounts for all
//  employees.
//
//  Input Parameters (passed by value):
//
//    NumEmp         number of employees to be processed
//
//  Input/Output Parameters (passed by reference):
//
//    Emp[]          array of employee records
//
void CalcData(EmpRec Emp[], BonusRec Bonus[], int NumEmp){

   const double SSRate    = 0.0683;       // SSI tax rate
   const double BonusRate = 1.15;         // bonus factor

   //////////////////////////////////////////////////////////
   //  Calculate the insurance fee, income tax, social security
   //  tax and net pay for each employee, storing the results in
   //  the relevant arrays:
   //
   for (int count = 0; count < NumEmp; count++) {

      if (EarnedBonus(Emp[count].Name, Bonus, NumEmp))
         Emp[count].GrossPay = BonusRate * Emp[count].GrossPay;

      Emp[count].InsFee = Insurance(Emp[count].InsPlan,
                                    Emp[count].Age);
      Emp[count].FIT    = FedTax(Emp[count].GrossPay);
      Emp[count].SSI    = SSRate * Emp[count].GrossPay;
      Emp[count].NetPay = Emp[count].GrossPay - Emp[count].FIT -
                          Emp[count].SSI - Emp[count].InsFee;
   }
}
```



```
////////////////////////////////////////////////////////////////
//  Insurance is used to calculate the proper insurance fee
//  for an employee.
//
//  Input Parameters (passed by value):
//
//   Plan   character 'B' or 'D', indicating plan selected
//   Age    employee's age
//
//  Return value:
//
//   InsFee    insurance fee charged to employee
//
float Insurance(char Plan, int Age) {

//  no changes from StructPay.cpp

}
////////////////////////////////////////////////////////////////
//  FedTax is used to calculate the proper income tax.
//
//  Input Parameter (passed by value):
//
//        Gross     employee's gross pay amount
//
//  Return value:
//
//        FIT       income tax withheld for employee
//
float FedTax(double Gross) {

//  no changes from StructPay.cpp

}
```



```
//////////////////////////////////////////////////////////////
//  EarnedBonus checks the list of bonus information to see if
//  the employee Emp should receive a bonus.
//
//  Input parameters:
//
//    Name[]      name of employee
//    Bonus[]     array of bonus information
//    NumEmp      number of employees
//
//  Returns true if bonus awarded, false if not.
//
bool EarnedBonus(const char Name[], const BonusRec Bonus[],
                 int NumEmp){

   for (int count = 0; count < NumEmp; count++) {

      if (strcmp(Name, Bonus[count].Name) == 0) {
         return Bonus[count].GetsBonus;
      }
   }
   return 0;
}
```



```
////////////////////////////////////////////////////////////
//  SortData sorts the array of NetPay amounts into increasing
//  order, rearranging the data in the other arrays to keep
//  everything consistent.  The selection sort algorithm is used.
//
//  Input Parameters (passed by value):
//
//    NumEmp      number of employees
//
//  Input/Output parameters (passed by reference):
//
//    Emp[]       array of employee records
//
void  SortData(EmpRec Emp[], int NumEmp) {

//  no changes from StructPay.cpp

}
```



```
//////////////////////////////////////////////////////////////
//  PrintTable is used to print the table of employee data to
//  output file.
//  Input Parameters (passed by value):
//    Emp[]       array of employee records
//    NumEmp      number of employees to print data for
//
void  PrintTable(const EmpRec Emp[], int NumEmp) {

  ofstream outPay;            // output file stream
  outPay.open("payola.dat");  // open the output file

                              // Set up for floating point output:
  outPay.setf(ios::fixed, ios::floatfield);
  outPay.setf(ios::showpoint);

  // Print the header for the table:
  PrintHeader(outPay);

  // Print the table, line by line:
  for (int count = 0; count < NumEmp; count++) {

     outPay << Emp[count].Name;
     int offset = 25 - strlen(Emp[count].Name);
     outPay << setw(offset) << Emp[count].IdNum;
     outPay << setw(12) << setprecision(2) << Emp[count].GrossPay;
     outPay << setw(10) << setprecision(2) << Emp[count].FIT;
     outPay << setw(10) << setprecision(2) << Emp[count].SSI;
     outPay << setw(10) << setprecision(2) << Emp[count].InsFee;
     outPay << setw(12) << setprecision(2) << Emp[count].NetPay;
     outPay << endl;
  }

  // Calculate and print averages:
  CalcAvgs(outPay, Emp, NumEmp);
}
```



```
////////////////////////////////////////////////////////////////
//  PrintHeader is used to print header for output file.
//
//  Output Parameter (passed by reference):
//
//        outPay         output file stream variable
//
void PrintHeader(ofstream& outPay) {

//  no changes from StructPay.cpp

}

////////////////////////////////////////////////////////////////
//  CalcAvgs is used to print final averages to input file.
//
//  Output Parameter (passed by reference):
//
//    outPay         output file stream variable
//
//  Input Parameters (passed by value):
//
//    Emp[]          array of employee records
//        NumEmployees   total number of employees
//
void CalcAvgs(ofstream& outPay, const EmpRec Emp[], int NumEmp) {

//  no changes from StructPay.cpp


}
```



```
////////////////////////////////////////////////////////////
//  InitRecords writes dummy values into the employee records,
//  prior to processing of input.
//
//  Output parameter:
//     Emp[]   array of employee records
//
void InitRecords(EmpRec Emp[]) {

   for (int idx = 0; idx < MaxEmp; idx++) {
      Emp[idx].IdNum    = 9999;
      strcpy(Emp[idx].Name, "Singh, Khan");
      Emp[idx].Age      = 150;
      Emp[idx].InsPlan  = 'X';
      Emp[idx].GrossPay = 0.0;
      Emp[idx].InsFee   = 0.0;
      Emp[idx].FIT      = 0.0;
      Emp[idx].SSI      = 0.0;
      Emp[idx].NetPay   = 0.0;

   }
}
```



```
//////////////////////////////////////////////////////////////
//  SkipWhiteSpace does just that --- reads the input file
//  stream ifile up to the next nonwhitespace character.
//
//  Input/Output parameter:
//    ifile    input file stream
//
void SkipWhiteSpace(ifstream& ifile) {

   char ch;

   ifile.get(ch);        // get first character

   while (ch == ' ' || ch == '\t' || ch == '\n') {
      ifile.get(ch);
   }

   ifile.putback(ch);    // last char read wasn't whitespace,
                         // so put it back
}
```


NamesPayS.cpp is a modification of NamesPayA.cpp to use Standard Library string objects rather than char arrays.

Accordingly, the new-style header files are incorporated.

Sample (second) input and output files:

```
Employee              Bonus?
Checkov, Pavel          Y
Chewbacca               N
Kirk, James             N
McCoy, Leonard          Y
Rand, Janice            Y
Riley, Kevin            N
Scott, Montgomery       Y
Skywalker, Luke         N
Solo, Han               N
Spock                   N
Sulu, Hikaru            N
Uhura                   Y
```

```
Programmer:  Bill McQuain
StarFleet Payroll


Name                 IdNum   Gross Pay      FIT       SSI       Ins       Net Pay
=============================================================================
Chewbacca             4297     1000.00    150.00     68.30    100.00       681.70
Rand, Janice          7077     1151.15    192.32     78.62     70.00       810.20
Skywalker, Luke       1716     2011.50    433.22    137.39    180.00      1260.89
Riley, Kevin          2334     2089.10    454.95    142.69    180.00      1311.47
Solo, Han             3424     2999.99    710.00    204.90    100.00      1985.09
Checkov, Pavel        5328     3450.00    858.50    235.63    180.00      2175.86
Sulu, Hikaru          3236     3897.89   1006.30    266.23     70.00      2555.36
Uhura                 7641     5409.12   1505.01    369.44    180.00      3354.67
Scott, Montgomery     5956     6032.23   1710.64    412.00    250.00      3659.59
Spock                 4098     8594.98   2556.34    587.04    140.00      5311.60
McCoy, Leonard        3175     8737.04   2603.22    596.74    100.00      5437.08
Kirk, James           5315     9336.00   2800.88    637.65    150.00      5747.47
=============================================================================
Average                       4559.08   1248.45    311.39    141.67      2857.58
```



```
//////////////////////////////////////////////////////////////
//  Title:         Payroll Program using C++ strings
//  Programmer:    Bill McQuain
//  ID Number:     000-00-0000
//  Compiler:      Microsoft Visual C++ ver 5.0
//  Platform:      Pentium II 400, Windows NT
//  Last modified: January 5, 1999
//
//  Purpose:   no changes from NamesPayA.cpp
//
//  Implementation Issues:
//
//    This implementation is an alteration of the char array
//    implementation (NamesPayA.cpp).  The only difference
//    is that this implementation uses the standard string
//    class instead of char arrays.
//
//    The first modification is in the #include section,
//    where new-style header files are used, and a using
//    statement is added to incorporate the std namespace.
//
//    In he global struct typedefs, the char array fields
//    are replaced by string fields.
//
//    Other modifications occur in the two input functions,
//    where string objects are used for input purposes, and
//    in the bonus decision function.
//
#include <fstream>
#include <iomanip>
#include <string>
using namespace std;

const int MaxEmp = 20;     // maximum number of employees
```



```
typedef struct {
   int     IdNum;          // employee's ID number
   string  Name;           // employee's name
   int     Age;            // employee's age
   char    InsPlan;        // employee's insurance plan code
   double  GrossPay;       // employee's gross pay
   double  InsFee;         // employee's insurance fee
   double  FIT;            // employee's federal income tax
   double  SSI;            // employee's social security tax
   double  NetPay;         // employee's net pay
} EmpRec;

typedef struct {
   string Name;            // employee name
   bool   GetsBonus;       // true if yes, false if no
} BonusRec;

//  Function prototypes:

void  InitRecords(EmpRec Emp[]);
void  ReadPayData(EmpRec Emp[], int& NumEmp);
void  ReadBonusList(BonusRec Bonus[]);
void  CalcData(EmpRec Emp[], BonusRec Bonus[], int NumEmp);
double FedTax(double EmpGrossPay);
double Insurance(char EmpInsPlan, int EmpAge);
bool  EarnedBonus(string Name, const BonusRec Bonus[], int NumEmp);
void  SortData(EmpRec Emp[], int NumEmp);
void  PrintTable(const EmpRec Emp[], int NumEmp);
void  PrintHeader(ofstream& outPay);
void  CalcAvgs(ofstream& outPay, const EmpRec Emp[], int NumEmp);
void  SkipWhiteSpace(ifstream& ifile);
```



```
//////////////////////////////////////////////////////////////////////
//  Beginning of main function:
//
int main() {

//  Local variables for main():

  EmpRec   Emp[MaxEmp];       // array of employee records
  BonusRec Bonus[MaxEmp];     // array of bonus records
  int       NumEmp;               // number of employees

// Initialize employee records:

  InitRecords(Emp);

// Read employee pay data and bonus list from input files:

  ReadPayData(Emp, NumEmp);
  ReadBonusList(Bonus);

// Calculate employee withholding data and net pay:

  CalcData(Emp, Bonus, NumEmp);

// Sort the data so net pay amounts are in increasing order:

  SortData(Emp, NumEmp);

// Print the table of employee information:

  PrintTable(Emp, NumEmp);

  return NumEmp;
}
```



```
//////////////////////////////////////////////////////////////////////
//  The function ReadPayData reads employee data from the input file a
//  stores it in the array parameters.  ReadData will read until the
//  end of the input file or until the arrays are filled (determined
//  by the global constant MaxEmp).
//
//  Output parameters (passed by reference):
//
//    Emp[]          array of employee records
//    NumEmp         number of employees for whom data was stored
//
void ReadPayData(EmpRec Emp[], int& NumEmp) {

  int   Index = 0;    // next array location to fill

  int    empnum,      // temp storage for employee id number
         empage;      //      employee age,
  double gross;       //      gross pay,
  char   plan;        //      insurance plan code, and
  string empname;     //      name

  ifstream inPay;                 // input file stream

  inPay.open("workers.dat");  // open input file

  // Read first data value from input file:

  inPay >> empnum;
```



```
  // Process and read until there's no more data or
  // we've filled up the arrays:

  while (inPay && Index < MaxEmp) {

     inPay >> empage >> gross >> plan;
     SkipWhiteSpace(inPay);      // skip whitespace after plan code
     getline(inPay, empname, '\n');

     Emp[Index].IdNum    = empnum;     // Store the last input line
     Emp[Index].Age      = empage;     //     in an employee record
     Emp[Index].GrossPay = gross;
     Emp[Index].InsPlan  = plan;
     Emp[Index].Name     = empname;

     inPay >> empnum;

     Index++;          // Update index for next store
  }

  NumEmp = Index;      // Remember: we never stored anything after
                       //           the last increment of Index

  inPay.close();       // Close the input file.
}
```



```
//////////////////////////////////////////////////////////////////
//  ReadBonusList reads list of employee names and flags indicating
//  whether each employee is to receive a bonus during this pay
//  period.
//  Output parameter:
//    Bonus[]     array of bonus records
//
void  ReadBonusList(BonusRec Bonus[]) {

   int    count = 0;          // number of data lines read
   string Name;               // employee name
   char   BFlag;              // bonus flag

   ifstream inBonus;
   inBonus.open("bonus.dat");
   inBonus.ignore(80, '\n');        // skip header line
                                    // Priming read:
   getline(inBonus, Name, '\t');    //   read name and
   inBonus >> BFlag;                //        flag
   inBonus.ignore(80, '\n');        //   skip to next line
   count++;                         //   count data line

   while (inBonus && count < MaxEmp) {

      Bonus[count].Name = Name;     // store bonus data
      Bonus[count].GetsBonus = (BFlag == 'Y');

      getline(inBonus, Name, '\t'); // repeat read logic
      inBonus >> BFlag;
      inBonus.ignore(80, '\n');
      count++;                      // count data line
   }
   inBonus.close();
}
```



```
////////////////////////////////////////////////////////////////
//  CalcData is used to calculate the net pay amounts for all
//  employees.
//  Input Parameters (passed by value):
//    NumEmp         number of employees to be processed
//  Input/Output Parameters (passed by reference):
//    Emp[]          array of employee records
//
void CalcData(EmpRec Emp[], BonusRec Bonus[], int NumEmp){

     // no changes from NamesPayA.cpp
}
////////////////////////////////////////////////////////////////
//  Insurance is used to calculate the proper insurance fee
//  for an employee.
//  Input Parameters (passed by value):
//   InsPlan   character 'B' or 'D', indicating plan selected
//   Age       employee's age
//  Return value:
//   InsFee    insurance fee charged to employee
//
float Insurance(char InsPlan, int Age) {

     //  no changes from StructPay.cpp

}
////////////////////////////////////////////////////////////////
//  FedTax is used to calculate the proper income tax.
//  Input Parameter (passed by value):
//        Gross     employee's gross pay amount
//  Return value:
//        FIT            income tax withheld for employee
//
float FedTax(double Gross) {

//  no changes from StructPay.cpp

}
```



```
////////////////////////////////////////////////////////////
//  EarnedBonus checks the list of bonus information to see if
//  the employee Emp should receive a bonus.
//
//  Input parameters:
//
//    Name        name of employee
//    Bonus[]     array of bonus information
//    NumEmp      number of employees
//
//  Returns true if bonus awarded, false if not.
//
bool EarnedBonus(string Name, const BonusRec Bonus[],
                 int NumEmp){

   for (int count = 0; count < NumEmp; count++) {

      if ( Name == Bonus[count].Name ) {
         return Bonus[count].GetsBonus;
      }
   }
   return 0;
}
```



```
////////////////////////////////////////////////////////////
//  SortData sorts the array of NetPay amounts into increasing
//  order, rearranging the data in the other arrays to keep
//  everything consistent.  The selection sort algorithm is used.
//
//  Input Parameters (passed by value):
//
//    NumEmp      number of employees
//
//  Input/Output parameters (passed by reference):
//
//    Emp[]       array of employee records
//
void  SortData(EmpRec Emp[], int NumEmp) {

//  no changes from StructPay.cpp

}
```



```
//////////////////////////////////////////////////////////////
//  PrintTable is used to print the table of employee data to
//  output file.
//  Input Parameters (passed by value):
//    Emp[]       array of employee records
//    NumEmp      number of employees to print data for
//
void  PrintTable(const EmpRec Emp[], int NumEmp) {

  ofstream outPay;            // output file stream
  outPay.open("payola.dat");  // open the output file
                              // Set up for floating point output:
  outPay.setf(ios::fixed, ios::floatfield);
  outPay.setf(ios::showpoint);

  // Print the header for the table:
  PrintHeader(outPay);

  // Print the table, line by line:
  for (int count = 0; count < NumEmp; count++) {

     outPay << Emp[count].Name;
     int offset = 25 - Emp[count].Name.length();
     outPay << setw(offset) << Emp[count].IdNum;
     outPay << setw(12) << setprecision(2) << Emp[count].GrossPay;
     outPay << setw(10) << setprecision(2) << Emp[count].FIT;
     outPay << setw(10) << setprecision(2) << Emp[count].SSI;
     outPay << setw(10) << setprecision(2) << Emp[count].InsFee;
     outPay << setw(12) << setprecision(2) << Emp[count].NetPay;
     outPay << endl;
  }

  // Calculate and print averages:
  CalcAvgs(outPay, Emp, NumEmp);
}
```



```
////////////////////////////////////////////////////////////////
//  PrintHeader is used to print header for output file.
//
//  Output Parameter (passed by reference):
//    outPay        output file stream variable
//
void PrintHeader(ofstream& outPay) {

     //  no changes from StructPay.cpp
}
////////////////////////////////////////////////////////////////
//  CalcAvgs is used to print final averages to input file.
//
//  Output Parameter (passed by reference):
//
//    outPay         output file stream variable
//
//  Input Parameters (passed by value):
//
//    Emp[]          array of employee records
//    NumEmployees   total number of employees
//
void CalcAvgs(ofstream& outPay, const EmpRec Emp[], int NumEmp) {

     //  no changes from StructPay.cpp
}
////////////////////////////////////////////////////////////////
//  SkipWhiteSpace does just that --- reads the input file
//  stream ifile up to the next nonwhitespace character.
//
//  Input/Output parameter:
//    ifile    input file stream
//
void SkipWhiteSpace(ifstream& ifile) {

     //  no changes from StructPay.cpp
}
```



```
////////////////////////////////////////////////////////////
//  InitRecords writes dummy values into the employee records,
//  prior to processing of input.
//
//  Output parameter:
//     Emp[]   array of employee records
//
void InitRecords(EmpRec Emp[]) {

   for (int idx = 0; idx < MaxEmp; idx++) {
      Emp[idx].IdNum    = 9999;
      Emp[idx].Name     = "Singh, Khan";
      Emp[idx].Age      = 150;
      Emp[idx].InsPlan  = 'X';
      Emp[idx].GrossPay = 0.0;
      Emp[idx].InsFee   = 0.0;
      Emp[idx].FIT      = 0.0;
      Emp[idx].SSI      = 0.0;
      Emp[idx].NetPay   = 0.0;

   }
}
```

