Structure, Array, and Pointer

Objectives

- Structures
- Arrays of Structures
- Structures as Function Arguments
- Dynamic Data Structure Allocation
- Problem Solving
- Common Programming Errors

Structures

- Historical holdovers from C
- **Record:** Provides for storing information of varying types under one name
- **Data field:** Individual data items in a record
- **Structure:** A record
- C++ permits methods to be included in a structure

Structures (continued)

- Structure creation and usage same as any variable
  - Declare and assign
- Declaration involves listing the data types, data names, and arrangement of data items
- **Populating the structure:** Assigning data values to data items
  - Structure members accessed by giving structure name and data item name
    - Separated by a period

Structures (continued)

```
int main()
{
   struct
   {
      int month;
      int day;
      int year;
   } birth;

   birth.month = 12;
   birth.day = 28;
   birth.year = 86;

   cout << "My birth date is "
        << birth.month << '/'
        << birth.day   << '/'
        << birth.year  << endl;

   return 0;
}
```

Structures (continued)

More frequently, the structure is declared as

```
struct birthNode {          // birthNode is a name for this structure
        int month;
        int day;
        int year;
        char name[20];
        long SSN;
        float hourly_rate;
};

birthNode birth; // birth is a var of type struct; similar to int a;

birth.month = 12;  //
birth.day = 28;
birth.year = 86;  birth.hourly_rate = 10.95;
```

Structures (continued)

- They can list form of structure with no following variable names
  - Defines new data type
    - Data structure of declared form
- Declaration may be global or local
- Initialization follows same rules as for initialization of arrays
- Any valid C++ data type can be used for individual data members

Structures (continued)

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

struct Date     // this is a global declaration
{
   int month;
   int day;
   int year;
};
```

Structures (continued)

```
int main()
{
  Date birth;

  birth.month = 12;
  birth.day = 28;
  birth.year = 86;

  cout << "My birth Date is " << birth.month << '/'
                      << birth.day   << '/'
                      << birth.year  << endl;

  return 0;
}
```

Arrays of Structures

- Power of structures realized when same structure used for lists of data
  - Integrity of data organization as a record can be maintained
- Declaring array of structures same as declaring array of any other type
- Once array of structures declared, data items accessed by giving position in array
  - Followed by period and appropriate structure member

Arrays of Structures (continued)

	Employee Number	Employee Name	Employee Pay Rate
1st structure →	32479	Abrams, B.	6.72
2nd structure →	33623	Bohm, P.	7.54
3rd structure →	34145	Donaldson, S.	5.56
4th structure →	35987	Ernst, T.	5.43
5th structure →	36203	Gwodz, K.	8.72
6th structure →	36417	Hanson, H.	7.64
7th structure →	37634	Monroe, G.	5.29
8th structure →	38321	Price, S.	9.67
9th structure →	39435	Robbins, L.	8.50
10th structure →	39567	Williams, B.	7.20

Arrays of Structures (continued)

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

struct PayRecord   // this is a global declaration
{
  int id;
  string name;
  double rate;
};
```

Arrays of Structures (continued)

```
int main()
{
  const int NUMRECS = 5;    // maximum number of records
  int i;
  PayRecord employee[NUMRECS] = {
                                { 32479, "Abrams, B.", 6.72 },
                                { 33623, "Bohm, P.", 7.54},
                                { 34145, "Donaldson, S.",  5.56},
                                { 35987, "Ernst, T.", 5.43 },
                                { 36203, "Gwodz, K.", 8.72 }
                                };

  cout << endl;    // start on a new line
  cout << setiosflags(ios::left);  // left justify the output
  for ( i = 0; i < NUMRECS; i++)
    cout << setw(7)  << employee[i].id
         << setw(15) << employee[i].name
         << setw(6)  << employee[i].rate << endl;

  return 0;
}
```

Structures as Function Arguments

- Individual structure members may be passed to a function
  - In same manner as scalar variables (such as int, float)
- Complete copies of all structure members can be passed to a function
  - Include name of structure as function argument
- Can pass structures by reference

Structures as Function Arguments (continued)

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

struct Employee          // declare a global structure type
{
  int idNum;
  double payRate;
  double hours;
};

double calcNet(Employee);  // function prototype
```

Structures as Function Arguments (continued)

```
int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;

  netPay = calcNet(emp);       // pass copies of the values in emp

    // set output formats
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);
```

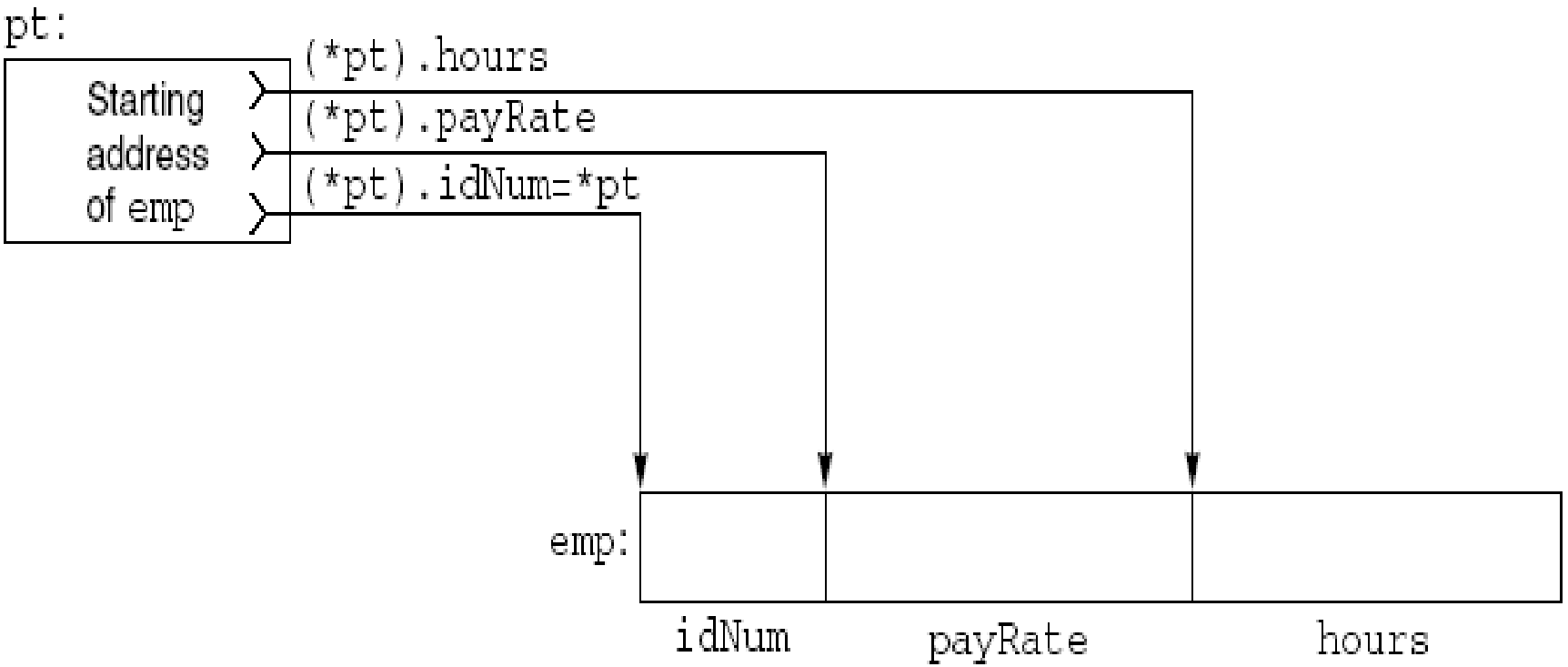
Structures as Function Arguments (continued)

```
  cout << "The net pay for employee " << emp.idNum
       << " is $" << netPay << endl;

  return 0;
}

double calcNet(Employee temp)  // temp is of data type Employee
{
  return (temp.payRate * temp.hours);
}
```

Passing a Pointer



A Pointer Can Be Used to Access Structure Members

Passing a Pointer (continued)

- Parentheses around expression **pt* necessary to access the structure whose address is in *pt*
- General expression *(*pointer).member* can be replaced with notation *pointer->member*
- Increment and decrement operators can be applied to individual structure data members

Passing a Pointer (continued)

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

struct Employee   // declare a global structure type
{
  int idNum;
  double payRate;
  double hours;
};
double calcNet(Employee *);   //function prototype
```

Passing a Pointer (continued)

```
int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;

  netPay = calcNet(&emp);       // pass an address

    // set output formats
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);
```

Passing a Pointer (continued)

```
  cout << "The net pay for employee " << emp.idNum
       << " is $" << netPay << endl;

  return 0;
}

double calcNet(Employee *pt) // pt is a pointer to a
{                            // structure of Employee type
  return (pt->payRate * pt->hours);
}
```

Passing a Pointer (continued)

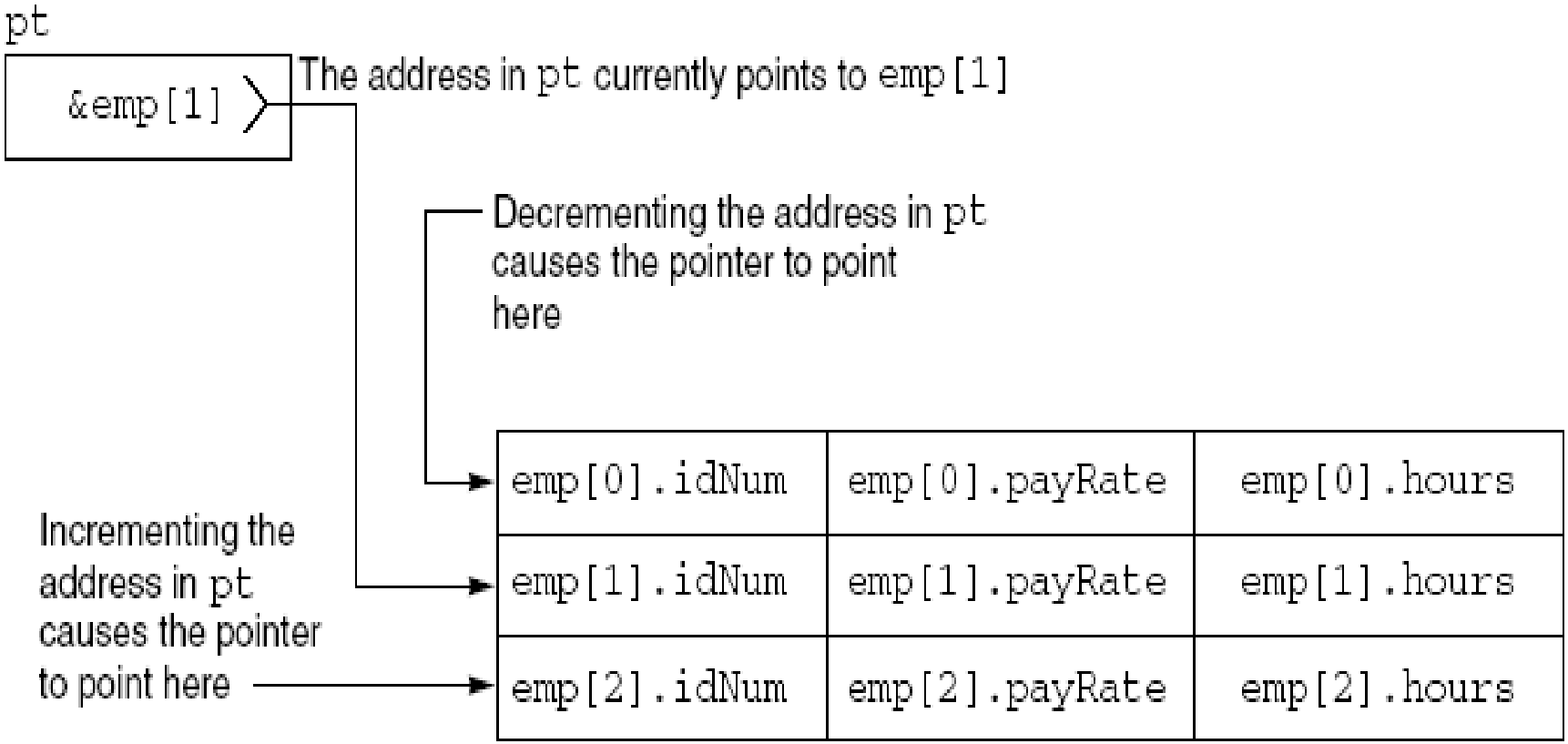


Figure: Changing Pointer Addresses

Function Argument vs. Passing a Pointer

double calcNet(Employee);

```
int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;

  netPay = calcNet(emp);       // pass copies

    // set output formats
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "The net pay for employee " << emp.idNum
       << " is $" << netPay << endl;

  return 0;
}

double calcNet(Employee temp)  // temp is of data t
{
  return (temp.payRate * temp.hours);
}
```

double calcNet(Employee *);

```
int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;

  netPay = calcNet(&emp);       // pass an addres

    // set output formats
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "The net pay for employee " << emp.idNum
       << " is $" << netPay << endl;

  return 0;
}

double calcNet(Employee *pt) // pt is a pointer to a
{                            // structure of Employee type
  return (pt->payRate * pt->hours);
}
```

Returning Structures

- Same procedures as for returning scalar values
  - Declaring function appropriately
  - Alerting calling functions to type of data structure being returned

Returning Structures (continued)

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

struct Employee   // declare a global structure type
{
  int idNum;
  double payRate;
  double hours;
};

Employee getValues();   // function prototype
```

Returning Structures (continued)

```
int main()
{
  Employee emp;

  emp = getValues();
  cout << "\nThe employee ID number is " << emp.idNum
       << "\nThe employee pay rate is $" << emp.payRate
       << "\nThe employee hours are " << emp.hours << endl;

  return 0;
}
```

Returning Structures (continued)

```
Employee getValues() // return an employee structure
{
  Employee next;

  next.idNum = 6789;
  next.payRate = 16.25;
  next.hours = 38.0;

  return next;
}
```

Dynamic Data Structure Allocation

- Permits list to expand as new records added and contract as records deleted
- User must provide *new* function with indication of amount of storage needed
  - Request enough space for particular type of data
  - No advance indication as to where computer will physically reserve requested number of bytes
  - No explicit name to access newly created storage locations

Dynamic Data Structure Allocation (continued)

- *new* returns address of first reserved location
  - Must be assigned to a pointer

Dynamic Data Structure Allocation (continued)

```
// a program illustrating dynamic structure allocation
#include <iostream>
#include <string>
using namespace std;

 struct TeleType
{
  string name;
  string phoneNo;
};

 void populate(TeleType *); // function prototype needed by main()
 void dispOne(TeleType *); // function prototype needed by main()
```

Dynamic Data Structure Allocation (continued)

```
int main()
{
  char key;
  TeleType *recPoint;     // recPoint is a pointer to a
                          // structure of type TeleType

  cout << "Do you wish to create a new record (respond with y or n): ";
  key = cin.get();
  if (key == 'y')
  {
    key = cin.get();      // get the Enter key in buffered input
    recPoint = new TeleType;
    populate(recPoint);
    dispOne(recPoint);
  }
  else
    cout << "\nNo record has been created.";

  return 0;
}
```

Dynamic Data Structure Allocation (continued)

```
// input a name and phone number
void populate(TeleType *record) // record is a pointer to a
{                               // structure of type TeleType
cout << "Enter a name: ";
getline(cin, record->name);
cout << "Enter the phone number: ";
getline(cin, record->phoneNo);

return;
}
```

Dynamic Data Structure Allocation (continued)

```
  // display the contents of one record

void dispOne(TeleType *contents)  // contents is a pointer to a
{                                 // structure of type TeleType
  cout << "\nThe contents of the record just created is:"
       << "\nName: " << contents->name
       << "\nPhone Number: " << contents->phoneNo << endl;

  return;
}
```

Focus on Problem Solving: Populating and Processing a Structure

Field No.	Field Contents	Field Type
1	Customer name	Character[50]
2	Customer address	Character[50]
3	Bicycles ordered	Integer
4	Bicycle type	Character—M or S
5	Creditworthiness	Character—Y or N
6	Dollar value of order	Float

Customer Record Layout

Focus on Problem Solving: Populating and Processing a Structure (continued)

Define the record

Function main

Populate the data using the function recvorder

Print the shipping instructions using the function shipslip

Function recvorder

Input data for name, address, number of bicycles, type of bicycle, and credit type

Calculate dollar value of order

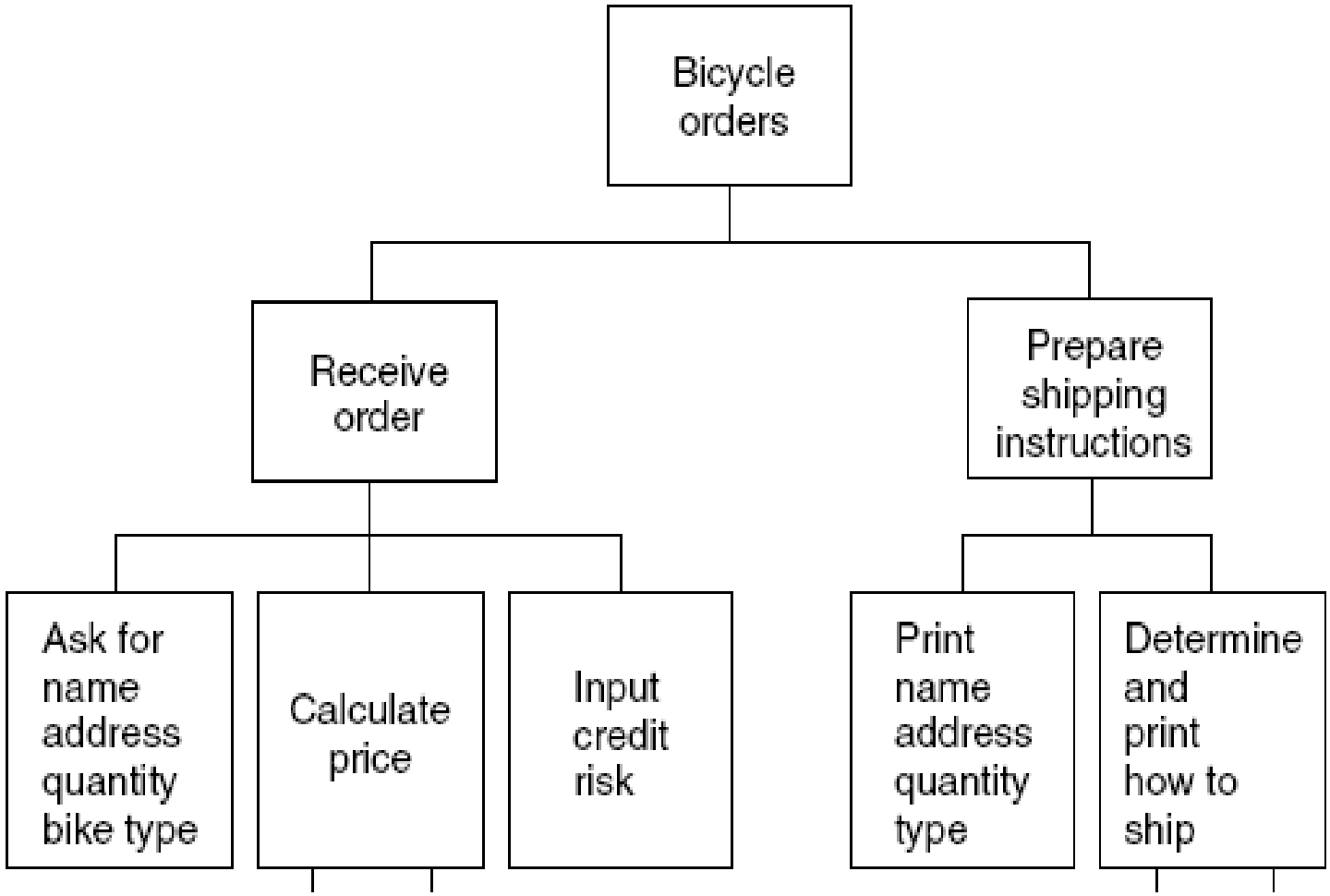
Function shipslip

Print name, address, number of bicycles, and type of bicycle

Determine if this is a C.O.D. or separate billable order

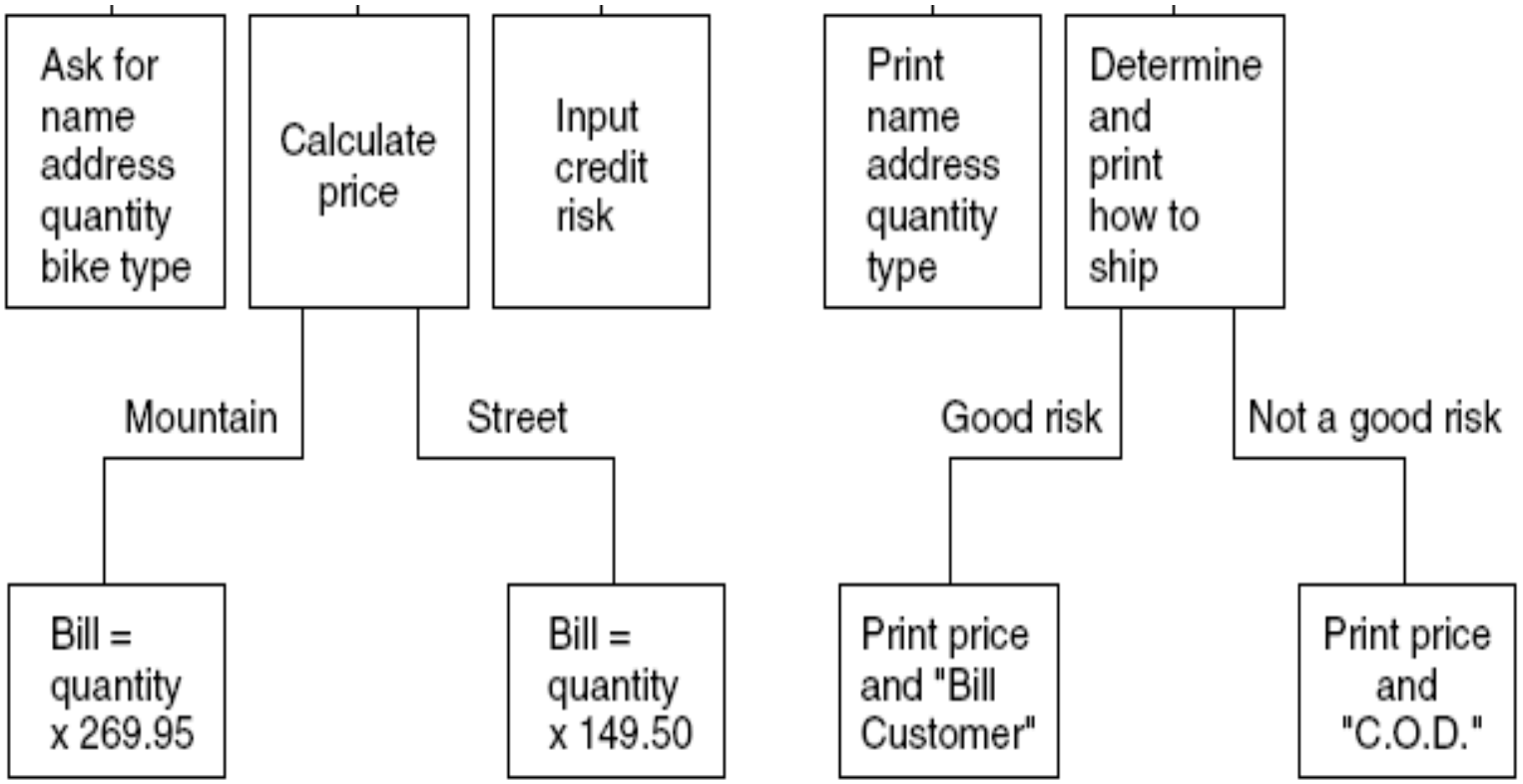
Print the order type and the dollar value of the order

Focus on Problem Solving: Populating and Processing a Structure (continued)



Structure Chart

Focus on Problem Solving: Populating and Processing a Structure (continued)



Focus on Problem Solving: Sorting and Searching an Array of Structures

- Arrays of records can be sorted and searched
- Entire record can be assigned to another record variable of same type
  - All fields within it are copied
- Usually interested in particular field
  - **Key field**

Focus on Problem Solving: Sorting and Searching an Array of Structures (continued)

Employee Number	Employee Name	Hourly Rate
34145	Donaldson, S.	5.56
33623	Bohm, P.	7.54
36203	Gwodz, K.	8.72
32479	Abrams, B.	6.72
35987	Ernst, T.	5.43

Focus on Problem Solving: Sorting and Searching an Array of Structures (continued)

Function main

Define an array of data structures and populate it

Sort the array of data structures using the function selsort

Display the sorted array using the function display

Prompt the user for an hourly rate and accept the data

Perform a linear search for records having a lower hourly rate and display the employee number and name for all records found

Function selsort

Perform a selection sort on the array based on the name field

Focus on Problem Solving: Sorting and Searching an Array of Structures (continued)

Function display

For each record in the array,

Display the record's contents

EndFor

Function linsearch

Search each record and examine its rate field

If the rate value is less than the input value,

Display the record's number and name fields

EndIf

Unions

- **Union:** Data type that reserves same area in memory for two or more variables
  - Can be different data types
- Only one type at a time can actually be assigned to union variable
- Union definition has same form as structure definition
  - Keyword *union* used in place of keyword *structure*
- Reserves sufficient memory locations to accommodate largest member's data type

Unions (continued)

- Same set of memory locations accessed by different variable names
  - Depending on data type of value currently residing in reserved location
- Each value stored overwrites previous value
- Individual union members accessed using same notation as structure members
- Unions may be members of structures or arrays
- Structures, arrays, and pointers may be members of unions

Common Programming Errors

- Structures and unions, as complete entities, cannot be used in relational expressions
- Using pointers to wrong data types when pointers reference data members of structures or unions
- Not keeping track of the currently stored variable in a union

Summary

- A *structure* allows individual variables to be grouped under a common variable name
- Each variable in a structure is accessed by its structure variable name, followed by a period, followed by its individual variable name
- A data type can be created from a structure
- Structures are useful as array elements

Summary (continued)

- Complete structures can be used as function arguments, in which case the called function receives a copy of each element in the structure
- The address of a structure may be passed to a function, either as a reference or a pointer
- Structure members can be any valid C++ data type, including other structures, unions, arrays, and pointers

Summary (continued)

- When a pointer is included as a structure member, a linked list can be created
- The definition of a union creates a memory overlay area, with each union member using the same memory storage locations