Structures

CS 1: Problem Solving & Program Design Using C++

Objectives

- Start on building structures
- Line up those arrays of structures
- See into using structures as function arguments
- Allocate structures dynamically
- Finally, look at common programming errors

Structures

- Arrays make it possible to access list of data of same data type using single variable name
- Need different mechanism to store information of varying types in one data structure
  - Mailing list data
  - Parts inventory data

Structures (2)

- STRUCTURE: data structure that stores different types of data under single name
  - A structure is a class that has no methods
  - All variables of class are public (default)
- Creating and using a structure requires two steps
  - Declaration
  - Assigning values

Form of a Structure

Name: Ronda Bronson-Karp

Street Address: 614 Freeman Street

City: Orange

State: NJ

Zip Code: 07052

Structure Declaration

- Lists the data types, data names and arrangements of data items
- EXAMPLE: Birth structure consists of three data items or fields called structure members

```
struct
{
        int month;
        int day;
        int year;
} birth;
```

Populating the Structure

- Assigning data values to data items of a structure
- Data structure members accessed by giving the structure name and individual data item name separated by period
  - Period called member access operator (dot operator)
- EXAMPLE: birth.month refers to the first member of birth structure
  - The following program assigns values to individual members of birth structure

Structures Example

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

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

Structures Example (2)

```
        birth.month = 12;
        birth.day = 28;
        birth.year = 1986;

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

        return 0;
}
```

About the Structures Example

- Output produced

  My Birth date is 12/28/1986

- Format of structure definition is flexible
  - The following is valid:

    struct {int month; int day; int year;} birth;

- Multiple variables can be declared in the same statement

  struct {int month; int day; int year;} birth, current;

- Creates two structures of the same form

  birth.day, birth.month, birth.year

  current.day, current.month, current.year

Modified Format for Defining Structures

```
struct Date
{
        int month;
        int day;
        int year;
};
```

- Date: structure type name that defines new data type that is a data structure of above form
- Definition statement: Date birth, current;
  - Reserves storage for two date structure variables named birth and current

Modified Structures Example

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

struct Date
{
        int month;
        int day;
        int year;
};

int main()
{
        Date birth;
```

Modified Structures Example (2)

```
        birth.month = 6;
        birth.day = 24;
        birth.year = 1982;

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

        return 0;
}
```

Initialization of Structures

- Follows same rules as initialization of arrays
  - Global and local structures initialized by following the definition with list of initializers
  - EXAMPLE: Date birth = {6, 24, 82} can replace first four statements in main() in the previous program
- Individual members of structure not limited to one data type
  - Can be any valid C++ data type including both arrays and structures

Arrays of Structures

- Real power of structures realized when same structure is used for lists of data
- Example: Process employee data
  - Option 1: consider each column as a separate list, stored in its own array
    - Employee numbers stored in array of integers
    - Names stored in array of strings
    - Pay rate stored in array of double-precision numbers
    - Correspondence between items for individual employee maintained by storing employee's data in same array position in each array

Arrays of Structures: List of Employee Data

Employee Number	Employee Name	Employee Pay Rate
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
36417	Hanson, H.	7.64
37634	Monroe, G.	5.29
38321	Price, S.	9.67
39435	Robbins, L.	8.50
39567	Williams, B.	7.20

Array of Structures

- Option 1 is not a good choice
  - Items related to single employee constitute a natural organization of data into structures
- Better option: use a structure
  - Data can be processed as single array of ten structures
  - Maintains integrity of the data organization as a record

Arrays of Structures: List of Employee Data Revisited

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

Declaring an Array of Structures

- Same as declaring an array of any other variable type
- If data type PayRecord is declared as:

```
struct PayRecord
{
        int idNum;
        string name;
        double rate:
};
```

- An array of ten such structures can be defined as:

```
PayRecord employee[10];
```

Referencing An Item in An Array of Structures

- `employee[0].rate` refers to rate member of the first employee structure in employee array
- Including structures as elements of array makes it possible to process list of structures using standard array programming techniques

Structures as Function Arguments

- Individual structure members are passed to a function in same manner as a scalar variable
- EXAMPLE: Given the structure definition

```
struct
{
        int idNum;
        double payRate;
        double hours;
} emp;
```

  - The statement display (emp.idNum); passes a copy of the structure member emp.idNum to a function named display()

Structures as Function Arguments (2)

- Passing complete copies of all members of structure to a function
  - Include name of structure as argument
- EXAMPLE: the function call `calcNet(emp);` passes a copy of the complete `emp` structure to `calcNet()`
  - A declaration must be made to receive structure

Structures as Function Arguments Example

- The following program declares a global data type for Employee structure
- This type is then used by both main() and calcNet() functions to define specific structures with names emp and temp
- The output produced by the program is:

The net pay for employee 6782 is $361.66

Structures as Function Arguments Code Example

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

struct Employee
{
        int idNum;
        double payRate;
        double hours;
};

double calculateNet (Employee);
```

Structures as Function Arguments Code Example (2)

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

        netPay = calculateNet (emp);

        cout << setw(10) << setiosflags(ios::fixed)
             << setiosflags(ios::showpoint) << setprecision(2);
```

Structures as Function Arguments Code Example (3)

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

        return 0;
}

double calculateNet(Employee temp)
{
        return (temp.payRate * temp.hours);
}
```

About the Structures as Function Arguments Example

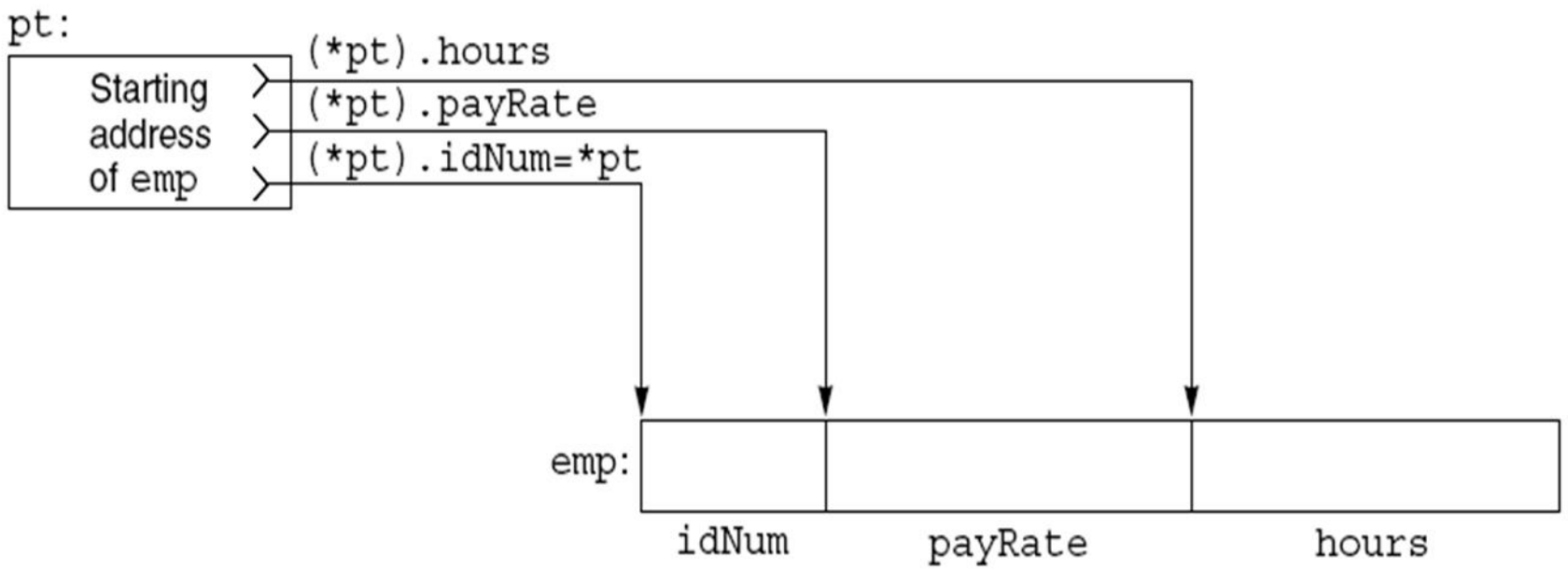
- Both main() and calcNet() use the Employee data type
- The variables defined in main() and calcNet() are different structures
- Changes to local temp variable in calcNet() are not reflected in emp variable of main()
- Same structure variable name could have been used in both functions with no ambiguity
- Both structure variables are local to their respective functions

Passing a Pointer

- Using a pointer requires modifications to the previous program:
  - Call to calcNet(): calcNet(&emp);
  - calcNet() function definition: calcNet(Employee *pt)
- Example:
  - Declares pt as a pointer to structure of type Employee
  - pt receives starting address of structure whenever calcNet() is called
  - calcNet() uses pt to access members of structure
  - (*pt).idNum refers to idNum member
  - (*pt).payRate refers to payRate member
  - (*pt).hours refers to hours member
  - These relationships illustrated in Figure 11.5
  - Parentheses around the expression *pt in Figure 11.5 are necessary to initially access “the structure whose address is in pt”

Passing a Pointer Visual

FIGURE 11.5 *A Pointer Can Be Used to Access Structure Members*



About the Passing a Pointer Visual

- Starting address of emp is also address of first member of the structure (Figure 11.5)
- Special notation commonly used
  - Expression (*pointer).member can always be replaced with notation pointer->member
- The following expressions are equivalent:
  - (*pt).idNum can be replaced by pt->idNum
  - (*pt).payRate can be replaced by pt->payRate
  - (*pt).hours can be replaced by pt->hours

Incrementing or Decrementing a Pointer

- Use increment or decrement operators
  - Expression ++pt->hours; adds one to hours member of emp structure
  - -> operator has higher priority than increment operator, therefore hours member is accessed first, then increment is applied
  - Expression (++pt)->hours; increments address before hours member is accessed
  - Expression (pt++)->hours; increments address after hours member is accessed

Passing a Pointer Code Example

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

struct Employee
{
        int idNum;
        double payRate;
        double hours;
};

double calculateNet (Employee *);
```

Passing a Pointer Code Example (2)

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

        netPay = calculateNet (&emp);

        cout << setw(10) << setiosflags(ios::fixed)
             << setiosflags(ios::showpoint) << setprecision(2);
```

Passing a Pointer Code Example (3)

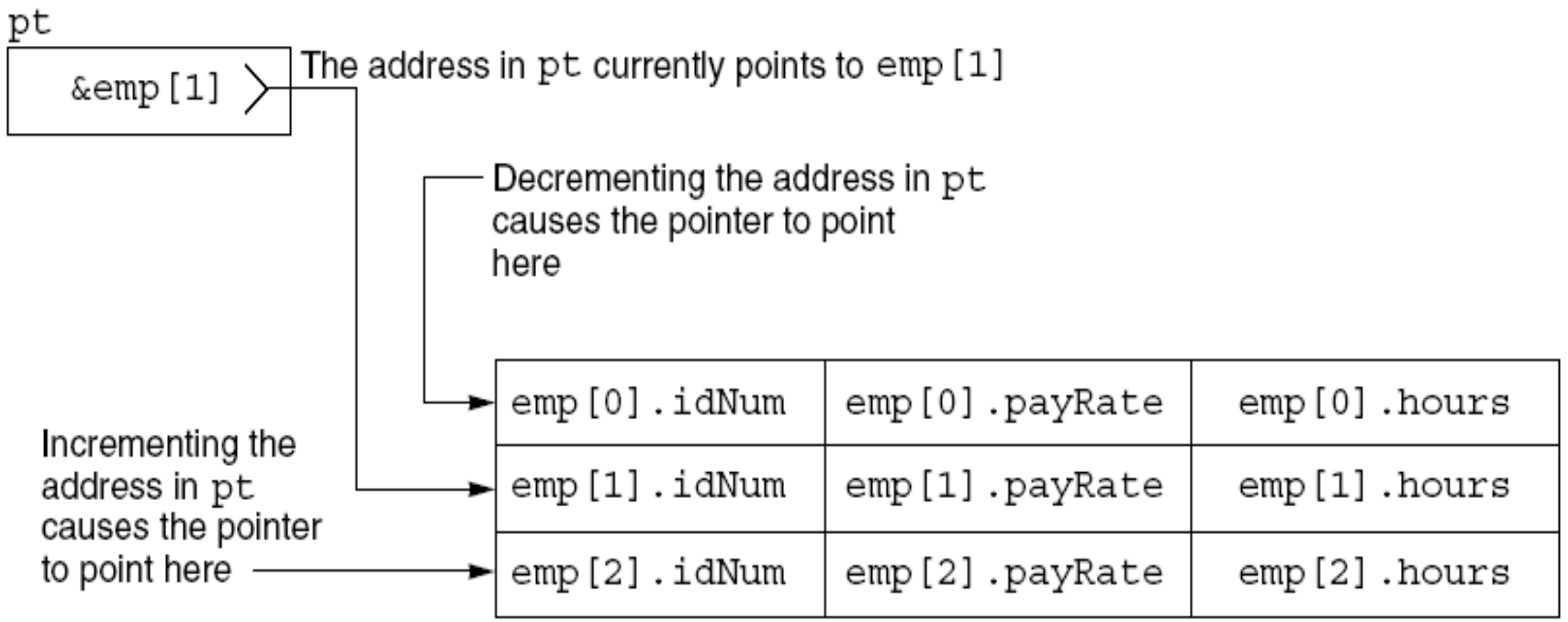
```
        cout << "The net pay for employee " << emp.idNum
                << " is $" << netPay << endl;

        return 0;
}

double calculateNet(Employee * temp)
{
        return (temp->payRate * temp->hours);
}
```

Passing a Pointer Example #2

- Array of three structures of type employee with address of emp[1] stored in the pointer variable pt
- Expression ++pt changes address in pt to starting address of emp[2]
- Expression --pt changes address in pt to starting address of emp[0]

Passing a Pointer Example #2 Visual

FIGURE 11.6 *Changing Pointer Addresses*



Returning Structures

- Structure handling functions receive direct access to structure by receiving structure reference or address
  - Equivalent to pass by reference
  - Changes to structure made in function
- Functions can also return separate structure
  - Must declare function appropriately and alert calling function to type of data being returned

Dynamic Structure Allocation

- Dynamic allocation of memory especially useful for lists of structures
  - Permits lists to expand and contract as records are added or deleted
- Additional storage space: Use new operator and indicate amount of storage needed
  - Expression new(int) or new int requests enough space to store integer number

Dynamic Structure Allocation (2)

Operator Name	Description
new	Reserves the number of bytes required by the requested data type; returns the address of the first reserved location or NULL if sufficient memory is NOT available
delete	Releases a block of bytes previously reserved; the address of the first reserved location is passed as an argument to the function

Dynamic Requesting Storage for a Structure

- If structure has been declared as follows:

```
struct TeleType
{
        string name;
        string phoneNo;
};
```

  - Expressions new(TeleType) or new TeleType reserve storage for one TeleType structure

Common Programming Errors

- Trying to use a structure, as a complete entity, in a relational expression
  - Individual members of structure can be compared, but not entire structure
- Using pointer in relation to structure and not indicating proper data type

Summary

- A structure allows individual variables to be grouped under common variable name
- A data type can be created from structure by using declaration form

```
struct Data-type
{
        individual member declarations;
};
```

- Individual structure variables may then be defined as this data type

Summary (2)

- Structures are useful as elements of arrays
- Complete structures can be function arguments
  - Called function receives copy of each element in structure
  - The address of structure may also be passed, either as reference or as pointer
- Structure members can be any C++ data type, including other structures, arrays, and pointers