Chapter 13: Structures

C++ FOR ENGINEERS AND SCIENTISTS

Objectives

- In this chapter you will learn about:
  - Single structures
  - Arrays of structures
  - Structures as function arguments
  - Linked lists
  - Dynamic data structure allocation
  - Unions
  - Common programming errors

Single Structures

- Creating and using a structure involves two steps
  - Declare record structure
  - Assign specific values to structure elements
- Declaring a structure requires listing data types, data names, and arrangement of data items
- Data items, or fields, are called **members** of a structure
- Assigning data values to members is called **populating the structure**

Single Structures (continued)

- An example structure definition:

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

Program 13.1

```
// A program that defines and populates a structure
#include <iostream>
using namespace std;

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;
}
```

Program 13.2

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

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

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

- The real power of structures is realized when the same structure is used for lists of data

Employee Number	Employee Name	Employee Pay Rate
32479	Abrams, B.	16.72
33623	Bohm, P.	17.54
34145	Donaldson, S.	15.56
35987	Ernst, T.	15.43
36203	Gwodz, K.	18.72
36417	Hanson, H.	17.64
37634	Monroe, G.	15.29
38321	Price, S.	19.67
39435	Robbins, L.	18.50
39567	Williams, B.	17.20

Figure 13.1 A list of employee data

Arrays of Structures (continued)

	Employee Number	Employee Name	Employee Pay Rate
1st structure →	32479	Abrams, B.	16.72
2nd structure →	33623	Bohm, P.	17.54
3rd structure →	34145	Donaldson, S.	15.56
4th structure →	35987	Ernst, T.	15.43
5th structure →	36203	Gwodz, K.	18.72
6th structure →	36417	Hanson, H.	17.64
7th structure →	37634	Monroe, G.	15.29
8th structure →	38321	Price, S.	19.67
9th structure →	39435	Robbins, L.	18.50
10th structure →	39567	Williams, B.	17.20

Figure 13.2 A list of structures

Program 13.3

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

const int NUMRECS = 5;   // maximum number of records

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

int main()
{
   int i;
   PayRec employee[NUMRECS] = {
                              { 32479, "Abrams, B.", 16.72},
                              { 33623, "Bohm, P.", 17.54},
                              { 34145, "Donaldson, S.", 15.56},
                              { 35987, "Ernst, T.", 15.43},
                              { 36203, "Gwodz, K.", 18.72}
                              };
   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;
}
```

Refer to page 737 for more explanations and examples

Structures as Function Arguments

- Structure members can be passed to a function just like any scalar variable
- Given the structure `emp` definition:

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

- Pass a copy of `emp.idNum` to `display()` function:

```
display(emp.idNum);
```

Structures as Function Arguments (continued)

- Copies of all structure members can be passed to a function by including the name of the structure as an argument to a called function
  - Example: `calcNet(emp)`

Program 13.4

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

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

double calcNet(Employee);  // function prototype

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);
   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;
 }
```

Program 13.4a

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

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

double calcNet(Employee&);       // function prototype

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

  netPay = calcNet(emp);         // pass a reference

// 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 a reference variable
{
  return temp.payRate * temp.hours;
}
```

Passing a Pointer

- Instead of passing references, pointers can be used
- Function call must take the address of the structure
  - Example: `calcNet(&emp);`
- Function declaration must indicate a pointer argument
  - Example: `calcNet(Employee *pt)`
- Inside function, `pt` argument is used to reference members directly
  - Example: `(*pt).idNum`

Passing a Pointer (continued)

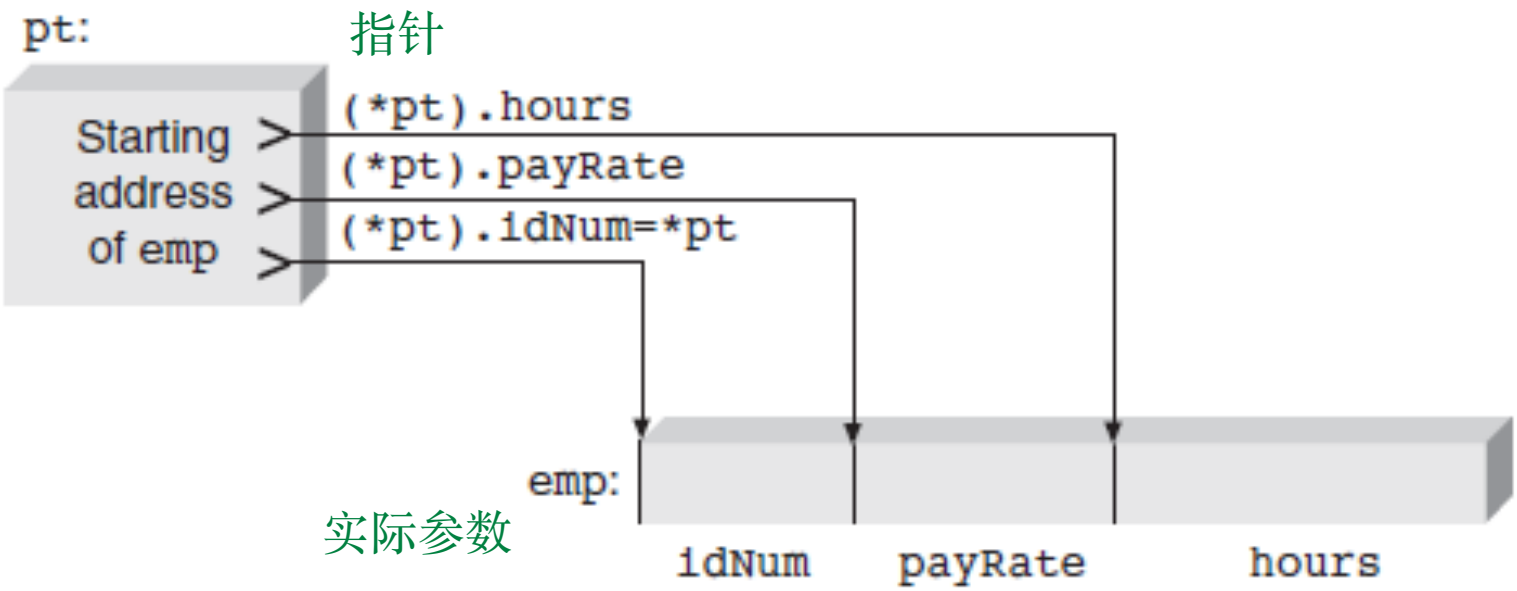


Figure 13.3 A pointer can be used to access structure members

Passing a Pointer (continued)

- Passing pointers to functions is very common
- Special notation exists for locating a member of a structure from a structure pointer
  - Notation is `pointer->member`
    - Equivalent to `(*pointer).member`
  - Examples:

```
(*pt).idNum      can be replaced by  pt->idNum
(*pt).payRate    can be replaced by  pt->payRate
(*pt).hours      can be replaced by  pt->hours
```

Program 13.5

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

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

double calcNet(Employee *);  //function prototype

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);
  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)

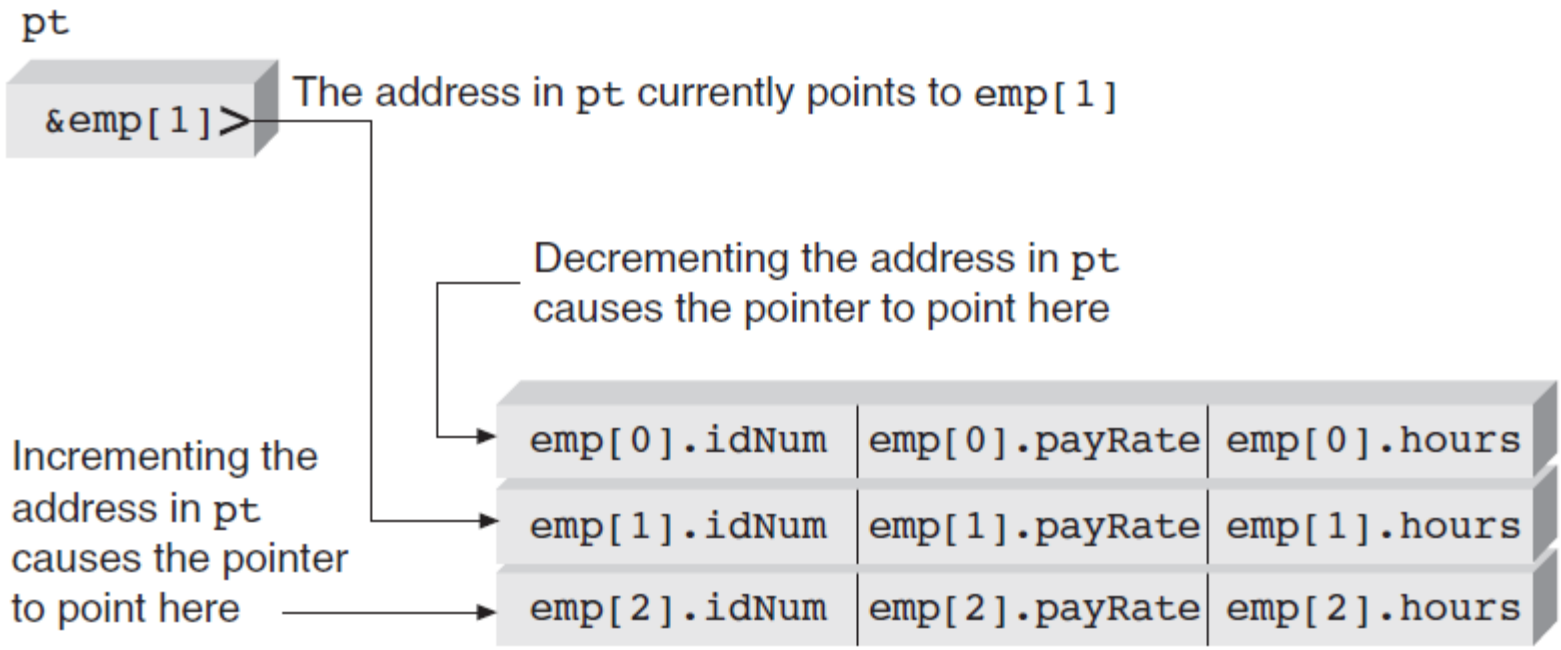


Figure 13.4 Changing pointer addresses

Refer to page 744 for more explanations and examples

Returning Structures

- In practice, most structure-handling functions get direct access to a structure by receiving a structure reference or address
  - Changes to a structure can be made directly from a function
- To have a function return a separate structure, follow same procedure as for returning scalar values

Program 13.6

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

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

Employee getVals();   // function prototype

int main()
{
  Employee emp;

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

  return 0;
}

Employee getVals()  // return an Employee structure
{
  Employee next;

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

  return next;
}
```

Linked Lists

- A classic data handling problem is making additions or deletions to existing structures that are maintained in a specific order
- Linked lists provide method for maintaining a constantly changing list
- **Linked list:** Set of structures in which each structure contains at least one member whose value is the address of next logically ordered structure in list

Refer to page 747 for more explanations and examples

Linked Lists (continued)

- Instead of requiring each record to be physically stored in correct order, each new structure is physically added wherever computer has free storage space
- Records are “linked” by including address of next record in the record immediately preceding it
- Current record contains address of next record no matter where next record is stored

Linked Lists (continued)

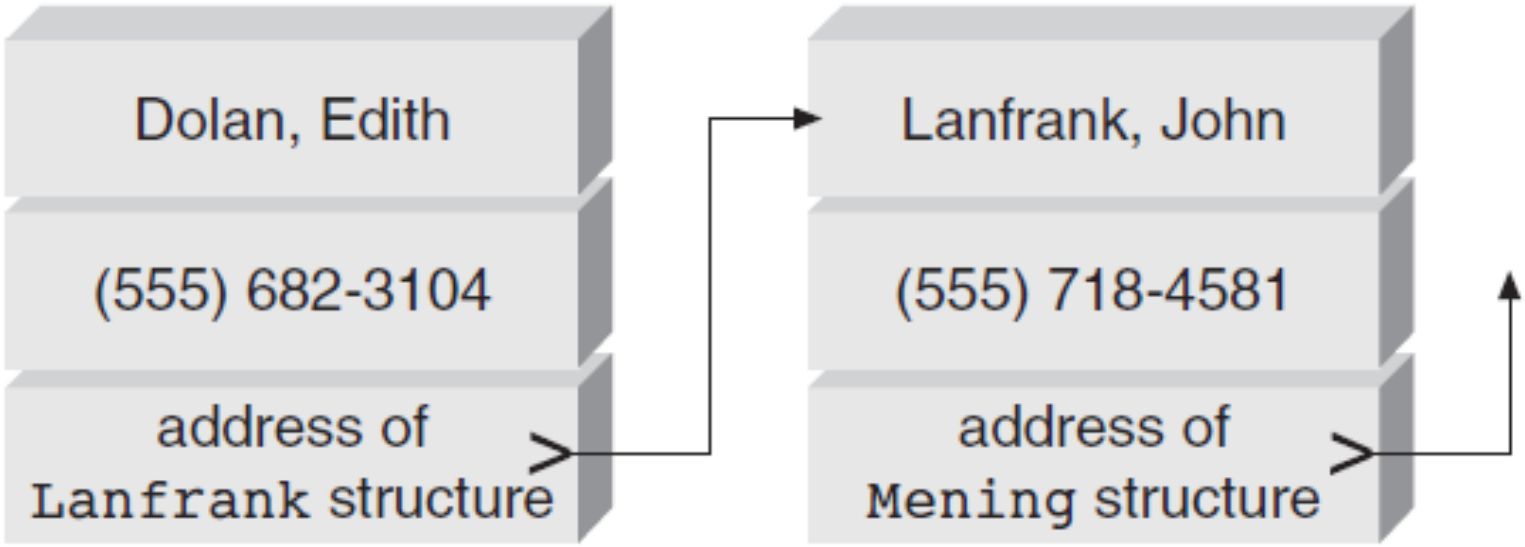


Figure 13.5 Using pointers to link structures

Linked Lists (continued)

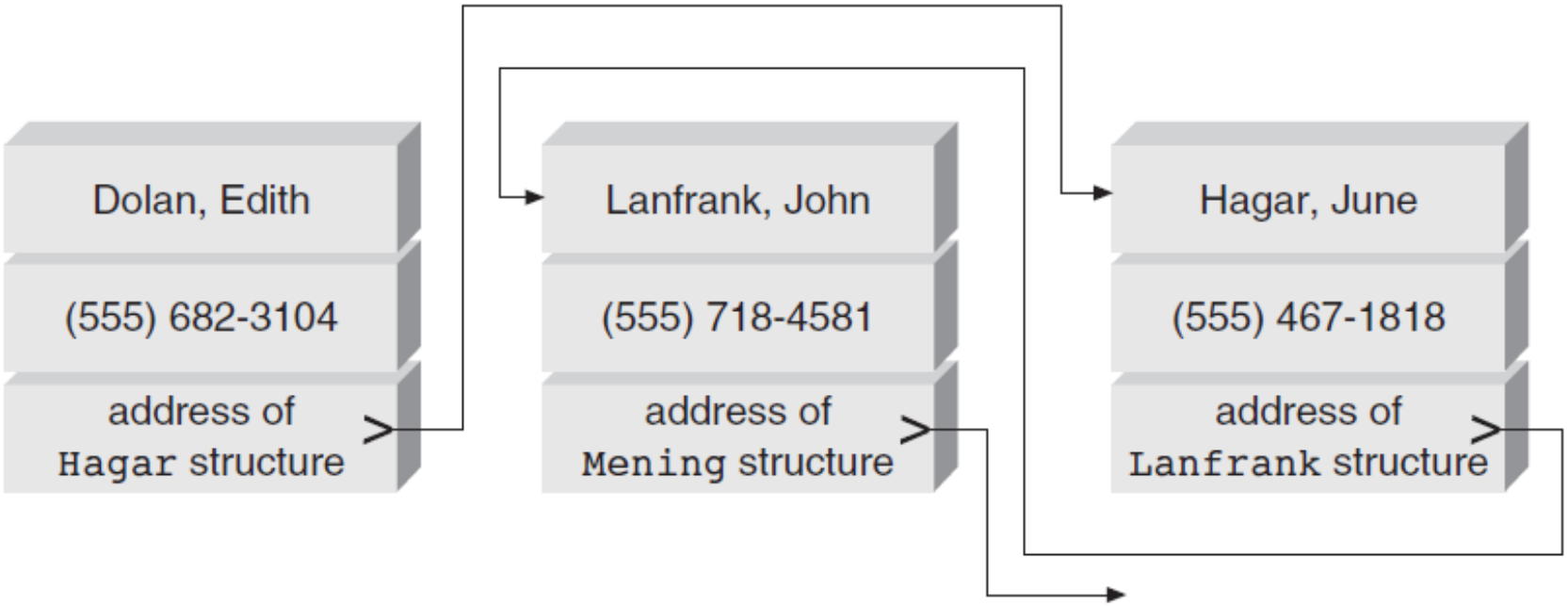


Figure 13.6 Adjusting addresses to point to the correct structures

Linked Lists (continued)

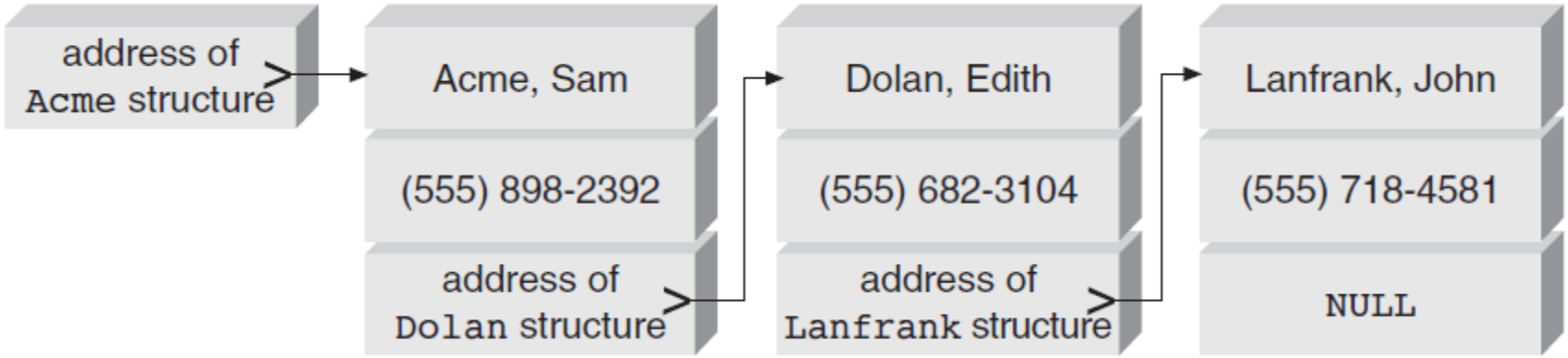


Figure 13.7 Using initial and final pointer values

Refer to page 749 for more explanations and examples

Linked Lists (continued)

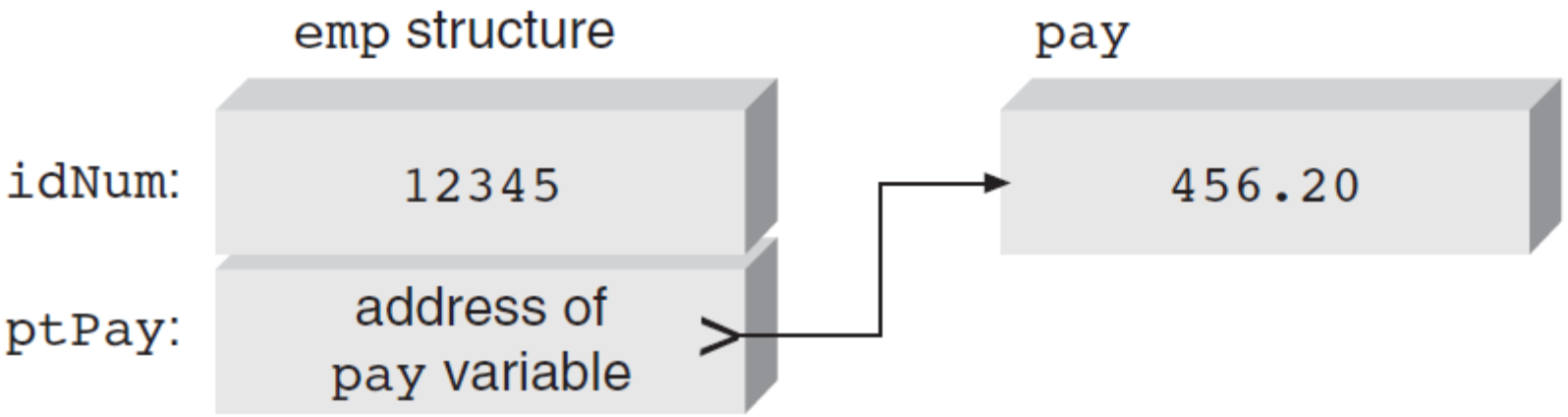


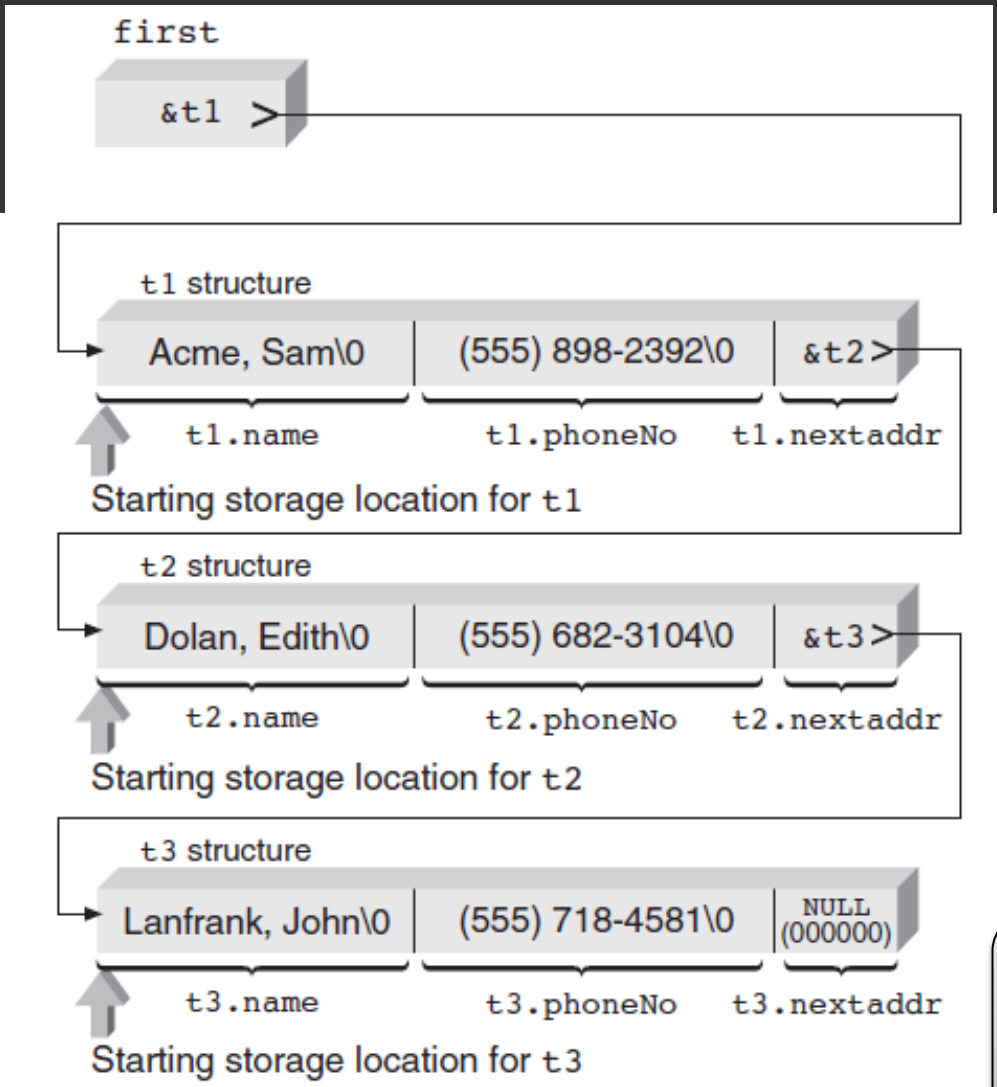
Figure 13.8 Storing an address in a structure member

Program 13.7

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

struct Test
{
   int idNum;
   double *ptPay;
};

int main()
{
   Test emp;
   double pay = 456.20;
   emp.idNum = 12345;
   emp.ptPay = &pay;

// Set output formats
   cout << setw(6)
        << setiosflags(ios::fixed)
        << setiosflags(ios::showpoint)
        << setprecision(2);
   cout << "\nEmployee number " << emp.idNum << " was paid $"
        << *emp.ptPay << endl;

   return 0;
}
```

Program 13.8

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

struct TeleType
{
  string name;
  string phoneNo;
  TeleType *nextaddr;
};

int main()
{
  TeleType t1 = {"Acme, Sam","(555) 898-2392"};
  TeleType t2 = {"Dolan, Edith","(555) 682-3104"};
  TeleType t3 = {"Lanfrank, John","(555) 718-4581"};
  TeleType *first;      // create a pointer to a structure

  first = &t1;          // store t1's address in first
  t1.nextaddr = &t2;    // store t2's address in t1.nextaddr
  t2.nextaddr = &t3;    // store t3's address in t2.nextaddr
  t3.nextaddr = NULL;   // store a NULL address in t3.nextaddr

  cout << endl << first->name
       << endl << t1.nextaddr->name
       << endl << t2.nextaddr->name
       << endl;

  return 0;
}
```



Refer to pages 754-755 for more explanations and examples

Figure 13.9 The relationship between structures in Program 13.8

Dynamic Structure Allocation

- Dynamic structure allocation uses the `new` and `delete` operators

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 not enough memory is available.
`delete`	Releases a block of bytes reserved previously. The address of the first reserved location is passed as an argument to the operator.

Table 13.1 Operators for Dynamic Allocation and Deallocation

Dynamic Structure Allocation (continued)

- Dynamic structure allocation permits expanding a list as new records are added and contracting the list as records are deleted
- When dynamically allocating storage, `new` operator must be provided with amount of space to allocate
- `new` returns a pointer to the storage just allocated

Refer to pages 757-760 for more explanations and examples

Unions

- **Union:** Data type that reserves same area in memory for two or more variables that can be different data types
- Definition of union has same form as structure definition
  - Keyword `union` replaces keyword `struct`

Unions (continued)

- Example: Union `val` contains a single member that can be character variable named `key`, integer named `num`, or double-precision variable named `volts`

```
union
{
   char key;
   int num;
   double volts;
} val;
```

Unions (continued)

- Example: Referencing union data member based on type indicator member

```
switch(uType)
{
  case 'c': cout << val.key;
            break;
  case 'i': cout << val.num;
            break;
  case 'd': cout << val.volts;
            break;
  default : cout << "Invalid type in uType: " << uType;
}
```

Unions (continued)

- Typically, second variable is used to keep track of current data type stored in union

```
struct
{
   char uType;
   union
   {
      char *text;
      float rate;
   } uTax;
}  flag;
```

Common Programming Errors

- Structures or unions cannot be used in relational expressions
- When pointer is used to point to a structure or a union, take care to use address in pointer to point to correct data type
- Be careful to keep track of the data stored in a union
  - Accessing data in a union by the wrong variable name is a troublesome error to locate

Summary

- Structure allows grouping variables under a common variable name
  - Each variable in structure accessed by structure name, followed by period, followed by variable name
- One form for declaring a structure is:

```
struct
{
   // member declarations
} structureName;
```

Summary (continued)

- A data type can be created from a structure by using this declaration form:

```
struct DataType
{
  // member declarations
};
```

Summary (continued)

- Structures are particularly useful as elements of arrays
  - Each structure becomes a record in a list of records
- Complete structures can be used as function arguments, in which case called function receives a copy of the structure elements
  - Structure addresses can also be passed as a reference or a pointer
  - When passed as pointer or reference, function has access to change original structure member values

Summary (continued)

- Structure members can be any valid C++ data type
  - May be other structures, unions, arrays, or pointers
- When pointer included in structure, a linked list can be created
- Unions are declared in same manner as structures
  - Definition of union creates a memory overlay area
  - Each union member uses the same memory storage locations

Homework

- P766, exercise 2
- P767, exercise 3