COMSATS University
Islamabad
(Lahore Campus)

CSC103- Programming Fundamentals

MS. MAHWISH WAQAS

MAWISH.WAQAS@CUILAHORE.EDU.PK

Chapter 9: Records (`structs`)

Objectives

- In this chapter, you will:
  - Learn about records (`structs`)
  - Examine various operations on a `struct`
  - Manipulate data using a `struct`
  - Learn about the relationship between a `struct` and functions
  - Examine the difference between arrays and `structs`

Objectives (cont'd.)

- Discover how arrays are used in a `struct`
- Learn how to create an array of `struct` items
- Learn how to create `struct`s within `struct`s

Introduction to Structures

Introduction to Structure

- A **structure** is a record/collection of data values, called *data members*, that form a single unit. E.g. a *student* struct may have id, name and marks data members.
- Unlike arrays, the data members can be of **different** types.
- C++ provides a new type (struct) for storing structures/records.

Records (`structs`)

- `struct`: a new type comprising a collection of a fixed number of components (members), accessed by name
  - Members may be of different types
- Syntax:

```
struct structName
{
    dataType1 identifier1;
    dataType2 identifier2;
       .
       .
       .
    dataTypen identifiern;
};
```

Records (`structs`) (cont'd.)

- A `struct` is a definition, not a declaration
  - Must declare a variable of that type to use it

```
struct houseType
{
    string style;
    int numOfBedrooms;
    int numOfBathrooms;
    int numOfCarsGarage;
    int yearBuilt;
    int finishedSquareFootage;
    double price;
    double tax;
};

  //variable declaration
houseType newHouse;
```

Records (`structs`) (cont'd.)

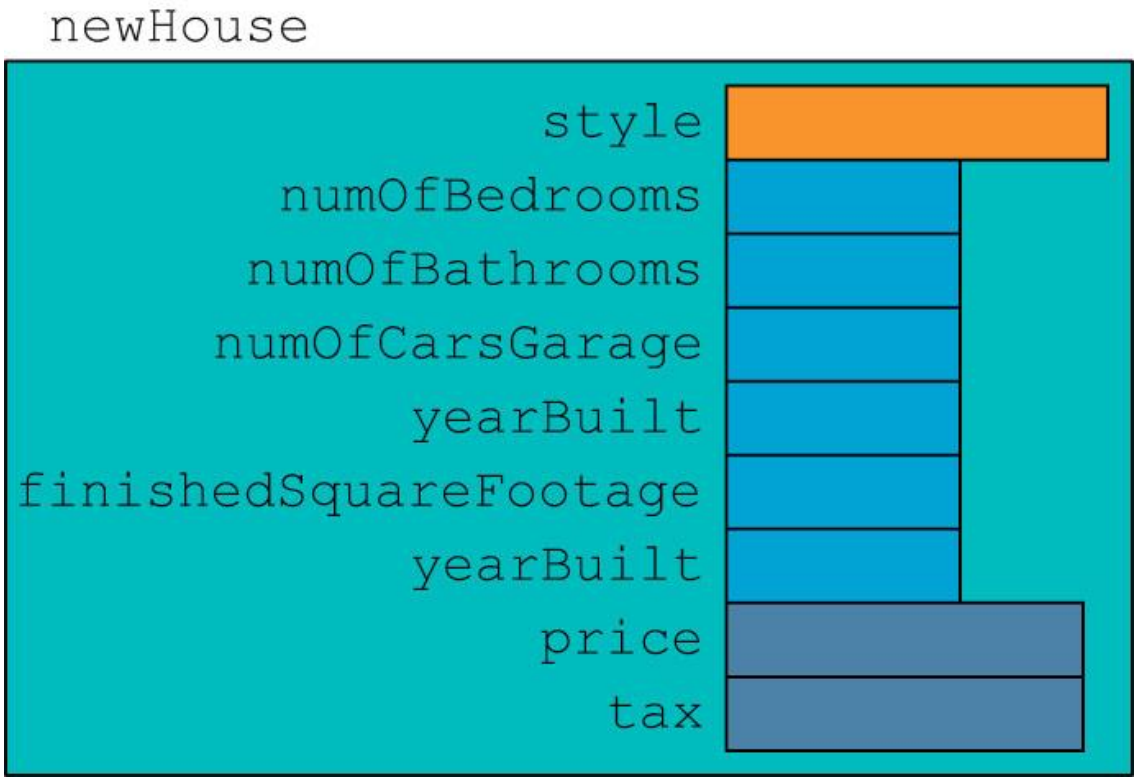


FIGURE 9-1 `struct` newHouse

Defining struct: Example 2

```
struct std_info {
char st_name[20];
char st_fam_name[20];
int id;  int grade;
};
```

- We define a new struct type **std_info**.
  - No memory is allocated, it is just definition.

```
std_info st1, st2, st3;
```

- We declare variables `st1,st2,st3` from **std_info**
  - Now, memory is allocated according to the size of data members

Naming conflict among data members

Members of the same structure type **must have unique names**, but two different structure types may contain members with same name without conflict.

```
struct employee
 {
        char Name[ 20 ];
        char Name[ 20 ]; // Error!!!
        int age;
        char gender;
        double hourlySalary;
};
```

```
struct Student
 {
        char Name[ 20 ]; // OK
        int age;
        char gender;
};
```

- The member names gender/Name in the employee struct have no conflict with that of the Student struct.

`struct` initialization

`struct` initialization

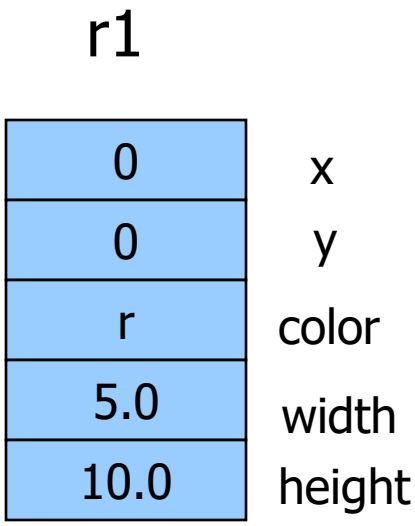
- Very similar to array initialization
- `std_info st1 = {"Ali", "Karimi", 9222, 10};`

- “Ali” is assigned to st_name
- “Karimi” is assigned to st_fam_name
- 9222 is assigned to id
- 10 is assigned to grade

- Order of values MUST be exactly the order of the members.
- The number of values must be <= the number of members. Initial values cannot be assigned in struct *definition*.

Example 2: Initializing Structures

```
struct Rect
{
    double x;
    double y;
    char color;
    double width;
    double height;
};
struct Rect r1 = {0,0,'r',5,10};
```



Structures as a New Data Type

- When we define a new **`struct`**, in fact we are defining a new *data type*.
  - Then, we use this new *data type* and define variables.
- So, we need to learn how do the following work for our new **`struct`** type?
  - Accessing members
  - Operators for **`struct`**
  - Array of **`struct`**
  - Pointer to **`struct`**
  - **`struct`** in functions

Accessing `struct` members

Accessing `struct` Members

- Syntax to access a `struct` member:

```
structVariableName.memberName
```

- The dot (.) is called the member access operator

Accessing `struct` Members (cont'd.)

- To initialize the members of `newStudent`:

```
newStudent.GPA = 0.0;
newStudent.firstName = "John";
newStudent.lastName = "Brown";
```

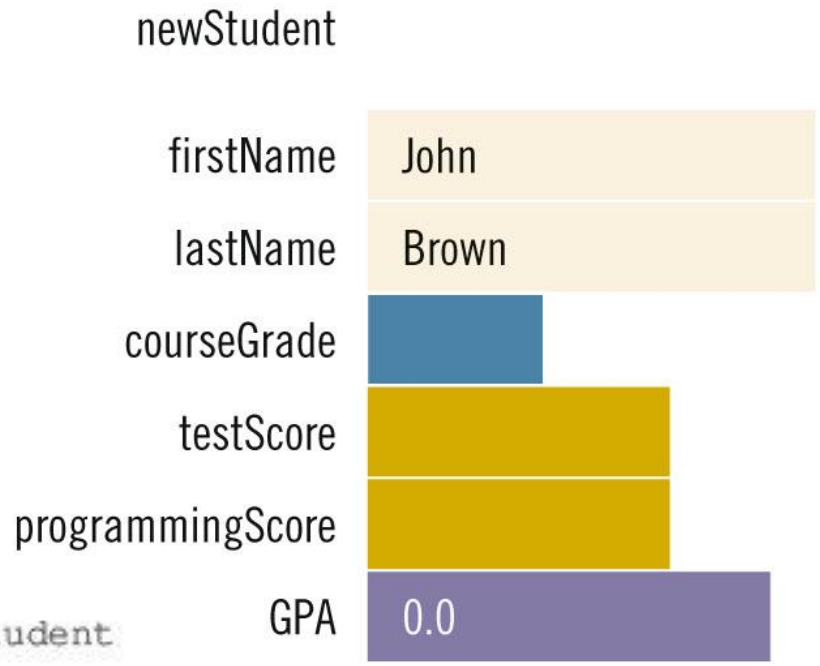


FIGURE 9-2 `struct` newStudent

Operations on `struct` variables

Assignment

- Value of one `struct` variable can be assigned to another `struct` variable of the same type using an assignment statement
- The statement:

```
student = newStudent;
```

copies the contents of `newStudent` into `student`

Assignment (cont'd.)

- The assignment statement:

```
student = newStudent;
```

is equivalent to the following statements:

```
student.firstName = newStudent.firstName;
student.lastName = newStudent.lastName;
student.courseGrade = newStudent.courseGrade;
student.testScore = newStudent.testScore;
student.programmingScore = newStudent.programmingScore;
student.GPA = newStudent.GPA;
```

Comparison (Relational Operators)

- Compare `struct` variables member-wise
  - No aggregate relational operations allowed
- To compare the values of `student` and `newStudent`:

```
if (student.firstName == newStudent.firstName &&
    student.lastName == newStudent.lastName)
.
.
.
```

Input/Output

- No aggregate input/output operations on a `struct` variable
- Data in a `struct` variable must be read or written one member at a time
- Example: output `newStudent` contents

```
cout << newStudent.firstName << " " << newStudent.lastName
     << " " << newStudent.courseGrade
     << " " << newStudent.testScore
     << " " << newStudent.programmingScore
     << " " << newStudent.GPA << endl;
```

`struct` and functions

`struct` Variables and Functions

- A `struct` variable can be passed as a parameter by value or by reference
- A function can return a value of type `struct`

```
void printStudent(studentType student)
{
    cout << student.firstName << " " << student.lastName
         << " " << student.courseGrade
         << " " << student.testScore
         << " " << student.programmingScore
         << " " << student.GPA << endl;
}
```

Class Demo

```
#include<iostream>
#include<cstring>
#include<string.h>
using namespace std;
struct Book
{
    int book_id;
    string book_name;
    float book_price;
};
void displayBook(Book b123);
```

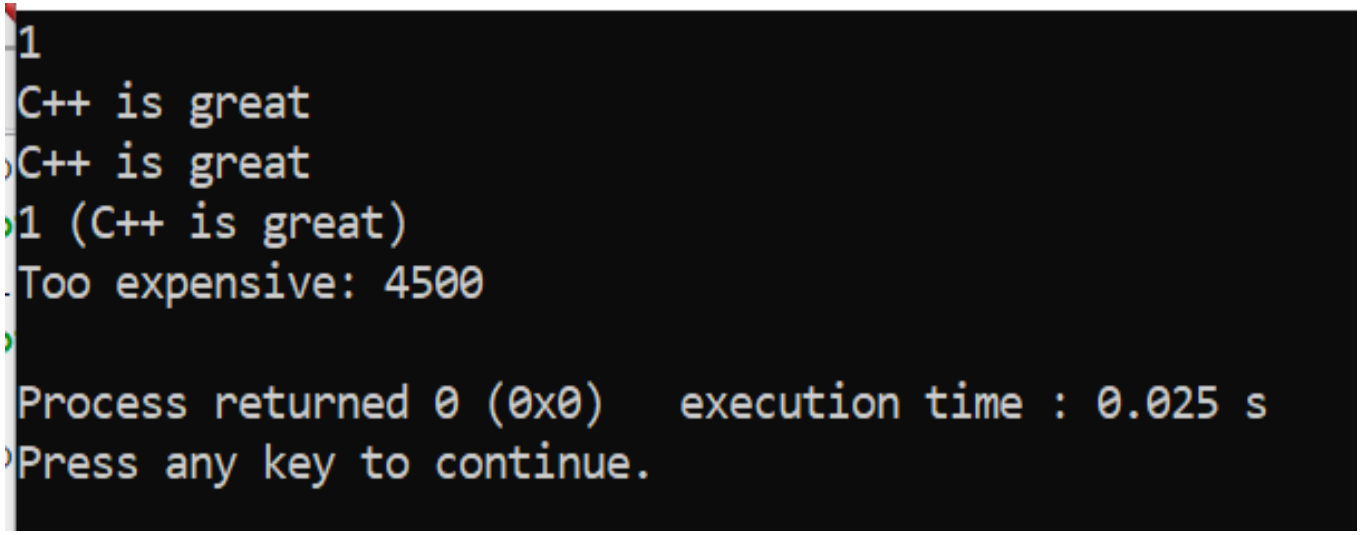
Class Demo (Contd.)

```
int main()
{

    Book b1 = {1, "C++ is best", 4500};
    cout<<b1.book_id<<endl;
    b1.book_name = "C++ is great";
    cout<<b1.book_name;

    Book b2 = b1;

    cout<<endl<<b2.book_name;

    if(b2.book_price > b1.book_price)
```

Class Demo (Contd.)

```
        cout<<"b2"<<endl;

    displayBook(b1);
    return 0;
}
void displayBook(Book b123)
{
    cout<<b123.book_id<< " ("<< b123.book_name<<") "
<<endl;
    if(b123.book_price>1000)
        cout<<"Too expensive: " <<b123.book_price<<endl;
    else
        cout<<"Too cheap: " <<b123.book_price<<endl;
}
```

Class Demo (Contd.)



Passing structs to functions

- Struct variables can be passed both by value and by reference.
  - Similar rules are followed as that of non-struct variables.
- Pass by value was demonstrated in the function displayBook previously.
  - See an example of pass by reference on the next slide.

Pass-by-reference Example

```
#include<iostream>
#include<cstring>
#include<string.h>
using namespace std;
struct Book
{
    int book_id;
    string book_name;
    float book_price;
};
void displayBook(Book
b123);
void updateBook(Book
&b123);
int main()
{
    Book b1 = {1, "C++ is
best", 4500};
    b1.book_name = "C++ is
great";
    Book b2 = b1;

    displayBook(b1);
    updateBook(b1);
    displayBook(b1);
    return 0;
}
```

Pass-by-reference Example (Contd.)

```
void displayBook(Book b123)
{
    cout<<b123.book_id<< "("<< b123.book_name<<")"<<endl;
    if(b123.book_price>1000)
        cout<<"Too expensive: " <<b123.book_price<<endl;
    else
        cout<<"Too cheap: " <<b123.book_price<<endl;
}
void updateBook(Book &b123)
{
    b123.book_price += 100;
    b123.book_name = b123.book_name + "!";
}
```

Pass-by-reference Example (Contd.)

```
1 (C++ is great)
Too expensive: 4500
1 (C++ is great!)
Too expensive: 4600

Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

Retruning structs from functions

- Struct variables can be returned from a function using the return statement.
  - Similar rules are followed as that of non-struct variables.
- See an example (the updateBook function) of returning a struct from a function on the next slide.
  - In this example, a local book variable is created in the updateBook function and returned back to the calling function (compare with the previous example)

Function returning struct Example

```
#include<iostream>
#include<cstring>
#include<string.h>
using namespace std;
struct Book
{
    int book_id;
    string book_name;
    float book_price;
};
void displayBook(Book b123);
Book updateBook(Book b123);
int main()
{
    Book b1 = {1, "C++ is best", 4500};
    b1.book_name = "C++ is great";

    displayBook(b1);
    Book b2 = updateBook(b1);

    displayBook(b2);
    return 0;
}
```

Function returning struct Example (Contd.)

```
void displayBook(Book b123)
{
    cout<<b123.book_id<< "("<< b123.book_name<<")"<<endl;
    if(b123.book_price>1000)
        cout<<"Too expensive: " <<b123.book_price<<endl;
    else
        cout<<"Too cheap: " <<b123.book_price<<endl;
}
Book updateBook(Book b123)
{
    b123.book_price += 100;
    b123.book_name = b123.book_name + "!";
    return b123;
}
```

Function returning struct Example (Contd.)

```
1(C++ is great)
Too expensive: 4500
1(C++ is great!)
Too expensive: 4600

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

`struct` and Arrays

Arrays versus `structs`

TABLE 9-1 Arrays vs. `structs`

Aggregate Operation	Array	`struct`
Arithmetic	No	No
Assignment	No	Yes
Input/output	No (except strings)	No
Comparison	No	No
Parameter passing	By reference only	By value or by reference
Function returning a value	No	Yes

Arrays in `structs`

- Two items are associated with a list:
  - Values (elements)
  - Length of the list
- Define a `struct` containing both items:

```
const int ARRAY_SIZE = 1000;

struct listType
{
    int listElem[ARRAY_SIZE];     //array containing the list
    int listLength;               //length of the list
};
```

Arrays in `structs` (cont'd.)

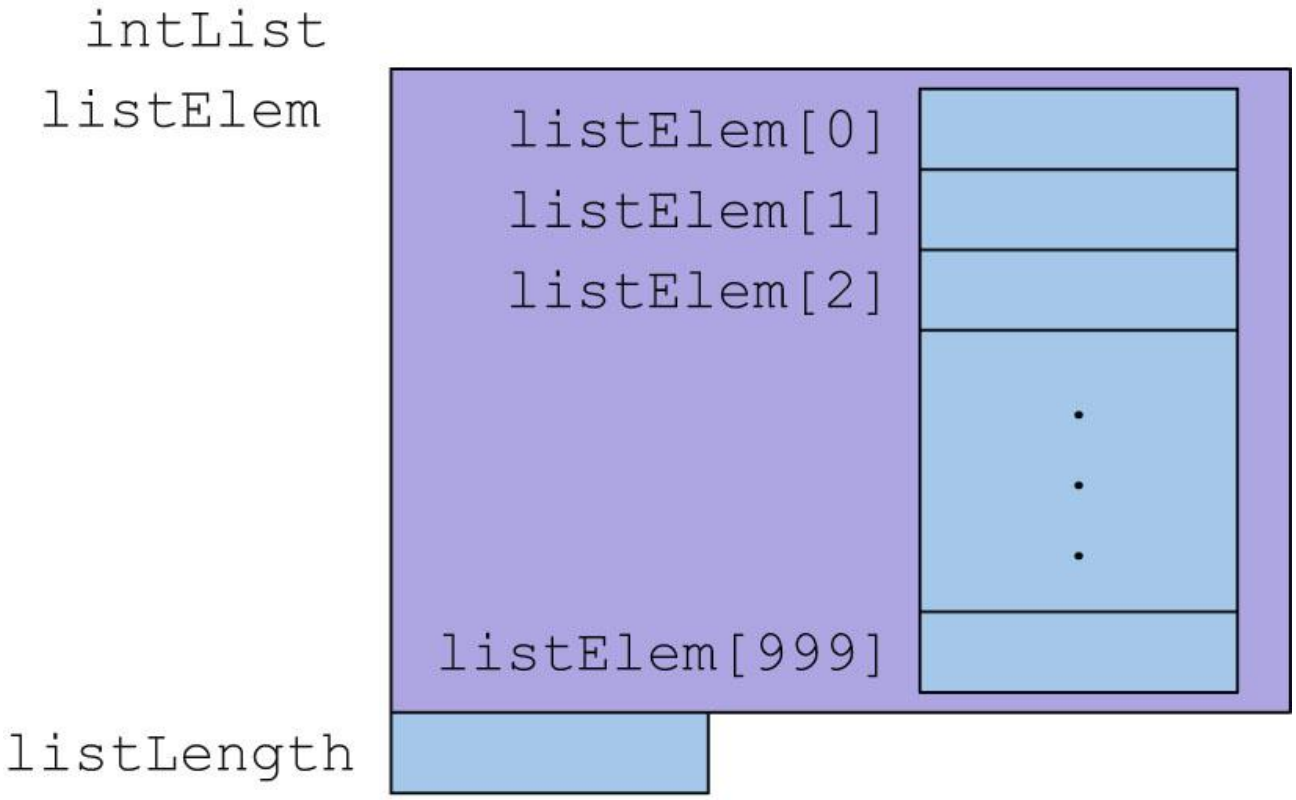
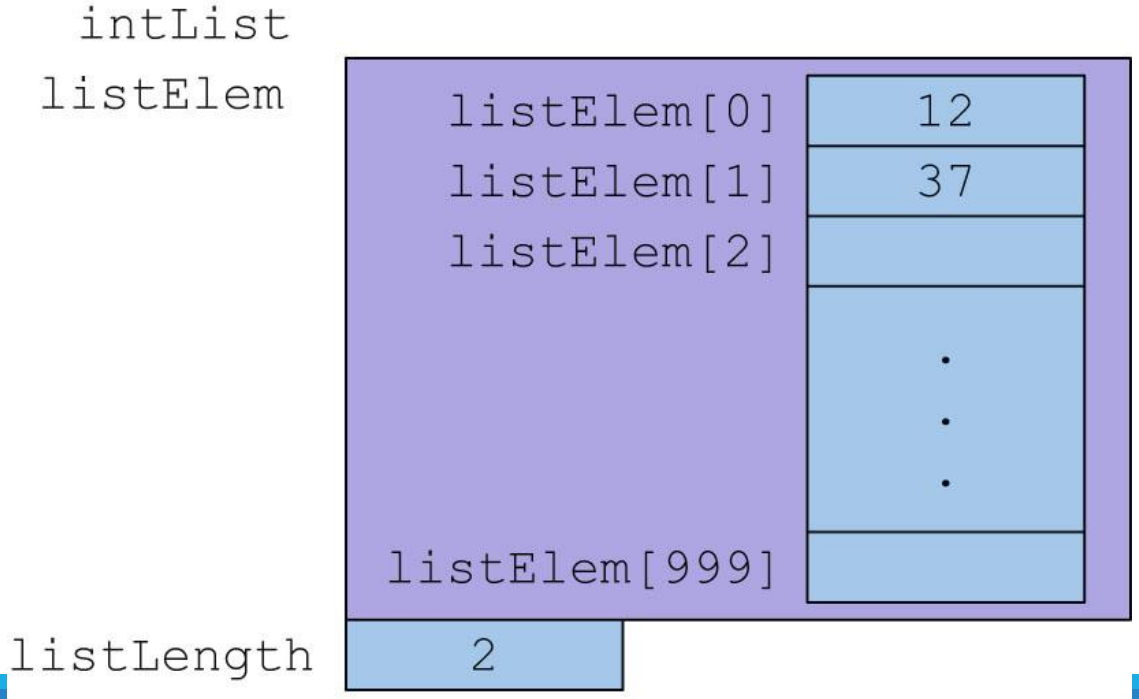


FIGURE 9-5 `struct` variable `intList`

Arrays in `structs` (cont'd.)

```
intList.listLength = 0;          //Line 1
intList.listElem[0] = 12;        //Line 2
intList.listLength++;            //Line 3
intList.listElem[1] = 37;        //Line 4
intList.listLength++;            //Line 5
```



Sequential (Linear) search

```
int seqSearch(const listType& list, int searchItem)
{
    int loc;

    bool found = false;

    for (loc = 0; loc < list.listLength; loc++)
        if (list.listElem[loc] == searchItem)
        {
            found = true;
            break;
        }

    if (found)
        return loc;
    else
        return -1;
}
```

structs in Arrays

- Example:

```
struct employeeType
{
    string firstName;
    string lastName;
    int    personID;
    string deptID;
    double yearlySalary;
    double monthlySalary;
    double yearToDatePaid;
    double monthlyBonus;
};
```

structs in Arrays (cont'd.)

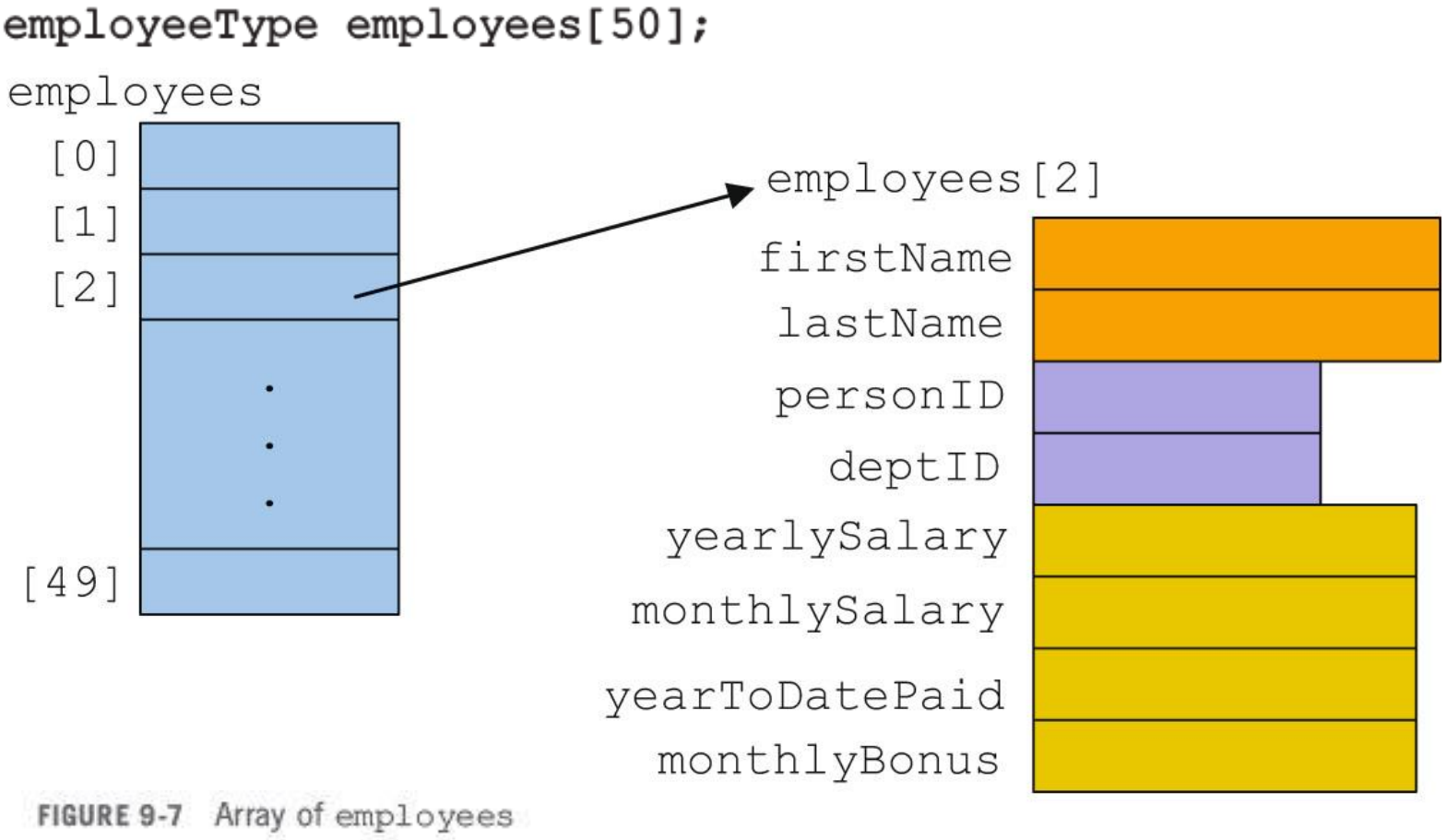


FIGURE 9-7 Array of employees

structs in Arrays (cont'd.)

- Example:

```
for (counter = 0; counter < 50; counter++)
{
  cin  >> employees[counter].firstName
  >> employees[counter].lastName
  >> employees[counter].personID
  >> employees[counter].deptID
  >> employees[counter].yearlySalary;

  employees[counter].monthlySalary =
  employees[counter].yearlySalary / 12;
  employees[counter].yearToDatePaid = 0.0;
  employees[counter].monthlyBonus = 0.0;
}
```

`structs` in Arrays (cont'd.)

- Suppose that for a given month, the monthly bonuses are already stored in each employee's record.
  - Now, we need to calculate the monthly paycheck and update the yearToDatePaid amount.
- The following loop computes and prints the employee's paycheck for the month:

structs in Arrays (cont'd.)

```
double payCheck; //variable to calculate the paycheck

for (counter = 0; counter < 50; counter++)
    cout << employees[counter].firstName << " "
         << employees[counter].lastName << " ";

    payCheck = employees[counter].monthlySalary +
               employees[counter].monthlyBonus;

    employees[counter].yearToDatePaid =
                          employees[counter].yearToDatePaid +
                          payCheck;

    cout << setprecision(2) << payCheck << endl;
}
```

Nested structs

A lot of members in struct

```
struct  employeeType
{
     string firstname;
     string middlename;
     string lastname;
     string empID;
     string address1;
     string address2;
     string city;
     string state;
     string zip;
     int hiremonth;
     int hireday;
     int hireyear;
     int quitmonth;
     int quitday;
     int quityear;
     string phone;
     string cellphone;
     string fax;
     string pager;
     string email;
     string deptID;
     double salary;
};
```

Solution: structs within a struct

These members can be further grouped into other struct types:

```
struct nameType
{
    string first;
    string middle;
    string last;
};

struct addressType
{
    string address1;
    string address2;
    string city;
    string state;
    string zip;
};
```

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

struct contactType
{
    string phone;
    string cellphone;
    string fax;
    string pager;
    string email;
};
```

Solution: structs within a struct

Now, we redefine the employeeType struct as:

```
struct employeeType
{
    nameType name;
    string empID;
    addressType address;
    dateType hireDate;
    dateType quitDate;
    contactType contact;
    string deptID;
    double salary;
};
```

Consider the following statement:

```
employeeType newEmployee;
```

This statement declares newEmployee to be a struct variable of type employeeType

structs within a struct

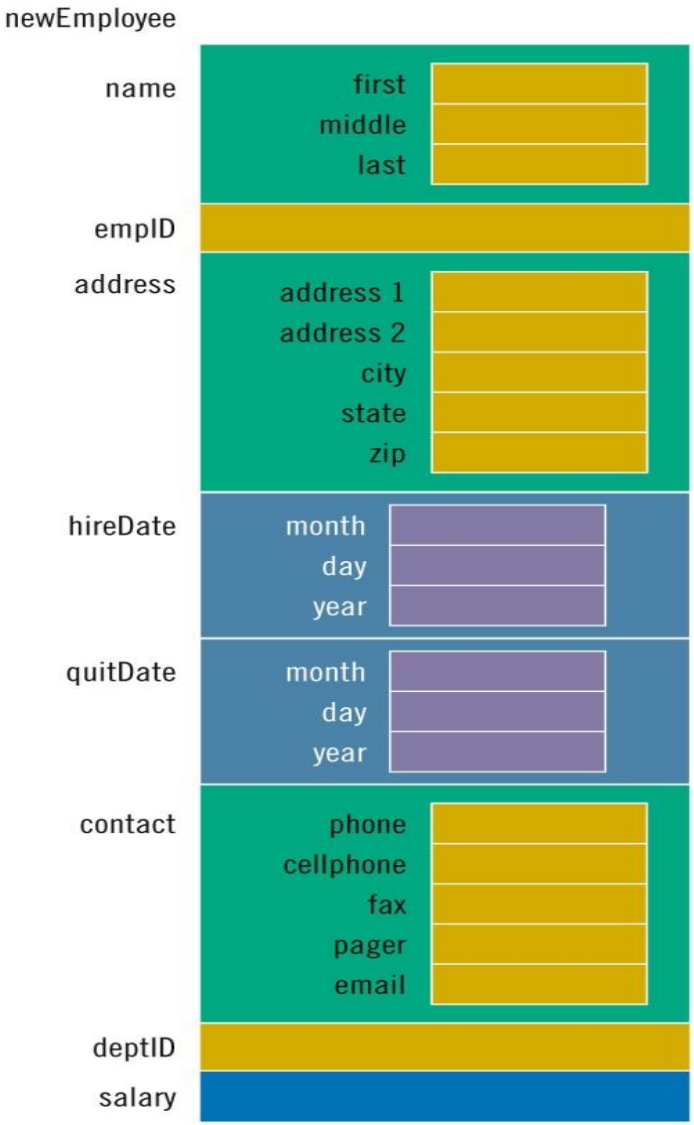


FIGURE 9-8 `struct` variable `newEmployee`

Using structs within a struct

- Use *cascaded* member access (i.e. multiple dot operators) to access a member nested at more than 1 level.

The statement:

```
newEmployee.salary = 45678.00;
```

sets the salary of newEmployee to 45678.00. The statements:

```
newEmployee.name.first = "Mary";
newEmployee.name.middle = "Beth";
newEmployee.name.last = "Simmons";
```

set the first, middle, and last name of newEmployee to "Mary", "Beth", and "Simmons", respectively.

Using structs within a struct

The statement:

```
cin >> newEmployee.name.first;
```

reads and stores a string into `newEmployee.name.first`. The statement:

```
newEmployee.salary = newEmployee.salary * 1.05;
```

updates the salary of `newEmployee`.

Using structs within a struct

The following statement declares `employees` to be an array of `100` components, wherein each component is of type `employeeType`:

```
employeeType employees[100];
```

The `for` loop:

```
for (int j = 0; j < 100; j++)
    cin >> employees[j].name.first >> employees[j].name.middle
        >> employees[j].name.last;
```

reads and stores the names of 100 employees in the array employees.

Pointers and Structs

- If you calculate the address of a struct
  - You get the address of the first byte in the struct (just like with ordinary variables)
  - Obviously, this is also the address of the first variable inside the struct
- Dereferencing a pointer to a struct gives you a reference to the struct
  - You would still use "." to access the individual variables inside the struct

The -> shortcut

- It's tedious to type (and hard to read) programs with lots of "(*p).x" stuff
- C/C++ provides an alternate syntax (that means precisely the same thing): p->x
- The -> is just a convenience, you can use either the (*p).x or p-> syntax, whatever you're most comfortable with
  - Most experienced programmers use the p->x

Pointer to struct Example

```
#include<iostream>
#include<cstring>
#include<string.h>
using namespace std;
struct Book
{
    int book_id;
    string book_name;
    float book_price;
};
void displayBook(Book *b123);
void updateBook(Book *b123);
int main()
{
    Book b1 = {1, "C++ is
best", 4500};
    b1.book_name = "C++ is
great";
    Book *bPtr = &b1;

    cout<<bPtr->book_id;
    displayBook(bPtr);
    updateBook(bPtr);
    displayBook(bPtr);
    return 0;
}
```

Pointer to struct Example
(Contd.)

```
void displayBook(Book
*b123)
{
    cout<<b123->book_id<<
"("<< b123-
>book_name<<")"<<endl;
    if(b123-
>book_price>1000)
        cout<<"Too
expensive: " <<b123-
>book_price<<endl;
    else
        cout<<"Too cheap:
" <<b123-
>book_price<<endl;
}

void updateBook(Book
*b123)
{
    b123->book_price +=
100;
    b123->book_name =
b123->book_name + "!";

}
```

Summary

- `struct`: collection of a fixed number of components
- Components can be of different types
  - Called members
  - Accessed by name
- `struct` is a reserved word
- No memory is allocated for a `struct`
  - Memory when variables are declared

Summary (cont'd.)

- Dot (.) operator: member access operator
  - Used to access members of a `struct`
- The only built-in operations on a `struct` are the assignment and member access
- Neither arithmetic nor relational operations are allowed on `structs`
- A `struct` can be passed by value or reference just like variables of other types
- A function can return a value of type `struct`
- `structs` can be members of other `structs`