Chapter 14

Structures

14.1 Structures

14.1.1 Basic Definitions
14.1.2 Examples of Structures
14.1.3 Accessing Structure Members
14.1.4 Arrays of Structures

14.2 Structures and Pointers

14.2.1 Assigning Structures
14.2.2 References to Structures
14.2.3 Pointers to Structures and the Arrow Operator

14.3 More Complex Structures

14.3.1 Arrays Within Structures

14.3.2 Structures Within Structures

14.3.3 Recursive Structures

14.3.4 Unions

14.3.5 Enumerations

14.3.6 New Names for Data Types

14.4 References

14.1 Structures

14.1.1 Basic Definitions

In C++, a **structure** is a collection of variables that are referenced under one name. This provides a convenient means of keeping related information together.

Structures are referred to as **compound data types**; they consist of several different variables, which are yet logically connected.

The general form of a structure declaration is:

```
struct struct_type_name {
                    type member_name_1;
                    type member_name_2;
                                    ...
                    type member_name_N;
} structure_variables;
```

The variables that comprise the structure are called **members**, **elements**, or **fields**.

14.1.2 Examples of Structures

- **Address**:

```
struct canadaAddress {
  char firstName[80];    // first name
  char lastName[80];     // last name
  char street[40];       // street name
  int  houseNumber;      // house number
  char city[40];         // city
  char province[3];      // province code
  char postalCode[7];    // postal code
} addressEntry;


canadaAddress addressVariable;
```

- **Vector**:

```
struct vector {
  float x;              // x coordinate
  float y;              // y coordinate
  float z;              // z coordinate
} referencePoint, leftUpper, lowerRight;
```

- **Employee**:

```
struct employee {
  char name[80];
  char phone[20];
  float hours;
  float wage;
};
employee databaseEntry;
```

14.1.3 Accessing Structure Members

Elements of a structure are accessed through the use of the **dot operator**, the general form of which is:

structure_variable . element_name

Example:

```
struct vector {
  float x;          // x coordinate
  float y;          // y coordinate
  float z;          // z coordinate
} referencePoint, leftUpper, lowerRight;

leftUpper.x = 0.5;
leftUpper.y = 1.3;
leftUpper.z = 7.9;
```

14.1.4 Arrays of Structures

It is common to use arrays of structures. However, the structure has to be defined first, before any array declarations that refer to this particular structure.

Example:

```
struct employee {
   char name[80];
   float hours;
   float wage;
};
employee staff[100];
```

Any entry in the database can be referred to by using the dot operator:

```
cout << staff[81].name;
staff[3].hours = 38.5;
```

14.2 Structures and Pointers

14.2.1 Assigning Structures

The contents of one structure can be assigned to another as long as both structures are **of the same type**.

```
struct mystruct { int a, b; };

int main()
{
  mystruct x, y;
  x.a = y.b = 10;  // svar1: 10  10
  x.a = y.b = 20;  // svar2: 20  20

  y = x; // assign structures
  return 0; }
```

14.2.2 References to Structures

A function can have a reference to a structure as a parameter or as a return type.

```
struct mystruct { int a; int b; };

mystruct &f(mystruct &var)
{
  var.a = var.a * var.a;
  var.b = var.b / var.b;
  return var;
}
void main()
{
  mystruct x, y;
  x.a = 10; x.b = 20;
  y = f(x); }
```

Any structures that have different type names are considered different by the compiler, even if the structure definitions look the same:

```
struct stypeA {
  int a, b;
};

struct stypeB {
  int a, b;
}

stypeA x;
stypeB y;

y = x; // Error: type mismatch
```

14.2.3 Pointers to Structures and the *Arrow* Operator

Structure pointers are declared as any other pointer variable, namely by putting an * in front of a structure´s variable name:

```
struct int_vector { int x, y, z; };
int_vector *int_vector_pointer;
```

To find the **address** of a structure variable, the & operator has to be placed before the structure variable´s name:

```
struct bal{
  float balance;
  char name[80];
} balance_record;

bal *rec; // a structure pointer to type bal

rec = &balance_record;
```

Accessing the Members of a Structure by Pointers

The members of a structure can be accessed through a pointer to the structure.

However, one **cannot** use the dot operator!

Instead, the **arrow operator** (->) has to be used. For example:

```
rec->balance
```

or

```
rec->name
```

Structure pointers are especially important as function parameters[1].

Pointers enable the passing of large structures as function arguments in an efficient and fast way.

1. See Chapters 14 and 15 for details.

14.3 More Complex Structures

14.3.1 Arrays Within Structures

A structure member that is an array is treated like any other data type.

```
struct stype {
  int numbers[10][10]; // 10 x 10 array of ints
  float b;
} var;
```

To reference integer 3,7 in **`numbers`** of **`var`** of structure **`stype`**, one would write:

```
var.numbers[3][7]
```

Note that the **array name** is indexed, not the structure name.

14.3.2 Structures Within Structures

A **nested structure** occurs when a structure is a member of a structure.

In the following example the structure **addr** is nested inside **emp**:

```
struct addr {
  char name[40];
  char street[40];
  char city[40];
  char zip[7];
};
struct emp {
  addr address;
  float wage;
} worker;

worker.address.zip = "T2N3F4";
```

14.3.3 Recursive Structures

A structure may also contain a pointer to a structure as a member.

The structure pointer can even point to the same structure type, which results in a **recursive** structure definition.

```
struct mystruct {
  int a;
  char str[80];
  mystruct *sptr; // pointer to mystruct object
};
```

Recursive structures are particularly useful for implementing **linked lists** for sorting and searching problems (see later for details), where data structures like the following are used:

```
struct int_list_entry {
   int value;
   int_list_entry *next_list_element;
};
```

14.3.4 Unions

In C++, a union is a memory location that is shared by two or more different variables. The union definition is similar to that of a structure, as the following example shows:

```
union utype {
  short int i;
  char ch;
} uvar;
```

This data type **`utype`** can hold either a short integer or a single character.

In uvar, both the short integer **`i`** and the character **`ch`** share the same memory location.

The compiler automatically allocates enough memory to hold the largest variable type in the union.

14.3.5 Enumerations

C++ allows to define a list of **named integer constants**.

Such a list is called an **enumeration**, which has the general format:

enum *enum_type_name* { ***enumeration_list*** } ***variable_list***;

The enumeration list represents the values a variable of the enumeration type can have.

```
enum apple {Jonathan, Golden_Del, Red_Del,
            Cortland, McIntosh} red, yellow;

apple fruit;
fruit = Cortland;
if(fruit == Red_Del)
  cout “Red Delicious\n”;
```

The key point about an enumeration is that each of the symbols stands for an **integer value**:

> The value of the first enumeration symbol is 0, the value of the second symbol is 1, etc.

Hence, enumeration symbols can be used in any integer expression.

```
cout << Jonathan << ' ' << Cortland;
```

Output: **0 4**

However, integers are not automatically converted to enumerated constants:

```
fruit = 1; // Error
```

It works only with a type cast:

```
fruit = (apple) 1; // OK, but poor style!!!
```

14.3.6 New Names for Data Types

C++ allows you to define a new name for an existing data type.

typedef *type* ***name***;

Here *type* is any valid data type, and ***name*** is the new name for this type.

This allows to use descriptive names for standard C++ data types or rename user-defined data types.

```
typedef float balance;

balance over_due = 123.56;
typedef apple pear;
```

Here **`balance`** and **`pear`** are just new names for the data types **`float`** and **`apple`**, respectively.

14.4 References

- H. Schildt, C++ from the Ground Up, McGraw-Hill, Berkeley, CA, 1998. Chapter 10.