# Structures

Definition: A structure is an aggregate data type which contains a fixed number of heterogeneous components.

- Structure components are termed fields or members, each with a unique name.
- Each field has a type, specified when the type is declared.

```
struct ItemType {
   int   Number;
   char  Name[25];
   float UnitPrice,
         CasePrice;
   char  AvailCode;
};
```

**struct structTypeName**
**{**
**// members**
**};**

**struct**
**{**
**// members**
**} structVar;**

**Not a variable declaration; NO storage is reserved until an instance of the structure type is declared.**

```
ItemType Item;
```

# Referencing Structure Fields

To reference a particular field, append a period (dot) and the field name to the structure variable:

The structure variable name acts as a qualifier signifying where the compiler should search first to locate the variable.

```
Item.UnitPrice

Item.Number
```


Fields of the same type can be listed in the same declaration, separated by a comma, (the same as normal variable declarations):

```
typedef struct {
   int   Number;
   char  Name[25];
   float UnitPrice,
         CasePrice;
   char  AvailCode;
} ItemType;


ItemType  Item;
```

**When structures are type defined the type name for the struct follows the delimiting brace, (following the typedef syntax, not the struct syntax).**

# Aggregate Operations

An operation on a data structure as a whole, as opposed to an operation on an individual component of the data structure. For example:

```
typedef struct {
    char  firstName[12];
    char  lastName[25];
    float gpa;
    int   quizGrade[10];
    int   finalExam;
} StudentRec;

StudentRec  x,y ;
```

| | | |
|---|---|---|
| Assignment: | `x = y;` | _____ |
| Comparison: | `x == y;` | _____ |
| I/O: | `cout << x;` | _____ |
| Arithmetic: | `x = x + y;` | _____ |
| Return: | `return x;` | _____ |
| Parameter: | `someFunc(x);` | _____ |

# Structure Field Access

Fields of a struct variable may be used just as if they were simple variables of the specified type.

Components are input just as simple variables of the same type:

```
cin >> Item.UnitPrice >> Item.AvailCode;
```

Elements of an array field are accessed the same as standard arrays:

```
float sumQuiz = 0.0f;
for (int i = 0;  i < 10; i++) {
   cout << x.quizGrade[i] << endl;
   sumQuiz += x.quizGrade[i];
}
cout << "Q Avg: " << sumQuiz/10.0f;
```

# Structure Manipulation

Structures can be accessed as whole entities:

Entire structures can be passed as parameters (default, by value):

```
void PrintItem (ItemType anItem ) {

   cout << anItem.Name
        << setw(28 - strlen(anItem.Name)) << anItem.Number
        << setw(10) << setprecision(2) << anItem.UnitPrice
        << setw(10) << setprecision(2) << anItem.CasePrice
        << "      " << anItem.AvailCode
        << endl;
}
```

Structure assignment copies corresponding field values from one structure instance to another (even array fields!):

```
void dupItem (const ItemType Source, ItemType& Target )
{
   Target = Source;
}
```

# Structure Comparison

However, structure comparison must be performed on an individual field basis:

```
bool ItemsAreEqual(const ItemType Item1, const ItemType Item2) {

   if (Item1.Number != Item2.Number)
      return false;

   if (Item1.UnitPrice != Item2.UnitPrice)
      return false;

   if (Item1.CasePrice != Item2.CasePrice)
      return false;

   if (Item1.AvailCode != Item2.AvailCode)
      return false;

   if (strcmp(Item1.Name, Item2.Name) != 0)  // most expensive
      return false;                          // comparison last

   return true;
}
```

# Structure Initialization

Structures can be initialized field by field:

```
ItemType  Item;
Item.Number    = 1001;
strcpy(Item.Name,“Instant Grits”);
Item.UnitPrice = 0.79;
Item.CasePrice = 5.99;
Item.AvailCode = ‘I’;
```

. . . or at declaration, initialized by a list (similar to arrays):

```
ItemType  Item = {1001,
                  “Instant Grits”,
                  0.79,
                  5.99,
                  ‘I’};
```

# Array of Structures

Array elements can be structures

```
typedef struct {
   int   Number;
   char  Name[25];
   float UnitPrice,
         CasePrice;
   char  AvailCode;
} ItemType;

ItemType  Items[50];
```

**Array of structures solves the parallel array problem.**

References:

```
Items[5].AvailCode          // availability code of 6th item
Items[7].UnitPrice          // unit price of 8th item
```

How much memory is the array `Items` array reserving?

# Array of Structures (continued)

The array of structures declared on the previous slide could be initialized from an input file by the following function:

```
bool ReadInventory(ifstream& In, ItemType Items[], const int MaxItems) {
   int NumItems, Idx = 0;
   In >> NumItems;
   if (NumItems > MaxItems) return false;   // too many items to store?
   while (In && Idx < NumItems) {
      In.ignore(80, '\n');
      In >> Items[Idx].Number;
      In.ignore(1, '\t');
      In.get(Items[Idx].Name, 25, '\t');
      In >> Items[Idx].UnitPrice >> Items[Idx].CasePrice;
      In >> Items[Idx].AvailCode;
      Idx++;
   }
   return (Idx == NumItems);              // wrong number of items found?
}
```

This design assumes the first line of the input file specifies the number of data lines that follow and that each subsequent data line lists field values in order, separated by tabs.

# Hierarchical Structures

Hierarchical structures are used when the data being represented exhibits nested relationships.

```
typedef struct {
   int  Number;
   char Name[25];
} ItemId ;
typedef struct {
   ItemId id;
   float UnitPrice,
         CasePrice;
   char  AvailCode;
} ItemType ;

ItemType Items[50];
```

**Note: ItemId must be declared before it is used in the declaration of the ItemType type.**

```
typedef struct {
   const int MaxItems = 1000;
   ItemType  Item[MaxItems];
   char StoreName[25];
   char StoreLocation[40];
} Inventory ;
```

```
Items[6].id.Number          // id number of the seventh item

Items[3].id.Name[7]         // eighth character of the name of the 4th item
```

# Unions

A union is a struct that holds only one of its members at a time during program execution. For example:

```
union WeightType {
  long   wtInOunces;
  int    wtInPounds;
  float  wtInTons;
};
WeightType weight;
```

At runtime, memory allocated to the variable `weight` does NOT include room for all three components. Instead, `weight` can contain only one of the following: either a long value or an int value or a float value.

The primary purpose of the union construction is to save memory. Because the compiler does not track which union field is currently selected, the use of unions is somewhat dangerous.