

STRUCTS AND UNIONS - INTRODUCTION

- ❑ Whereas an array is a collective type of like elements, a **struct** is a collective type of arbitrary elements. A **struct** is defined as follows

Usage:

```
struct unique_name {member_list};
```

- ❑ Each struct must have a unique name. This name is a new type that has been defined by the programmer
- ❑ The semi-colon after the last brace is essential to the definition
- ❑ member_list denotes the **field(s)** of the struct

STRUCT EXAMPLE

- ❑ Imagine a C++ program that is used to keep track of customers bank details. A suitable struct for such an implementation may be

```
struct bank_record  
{  
    int acc_number;  
    int date_of_birth;  
    float current_balance;  
};
```

The unique_name **bank_record** is now defined as a new type and can be used accordingly

MORE ON STRUCTS

- Variables of struct type can be declared in exactly the same way as variables of a fundamental type. So to declare a **new instance** of the struct `bank_record`

```
bank_record my_record;
```

which defines an instance of `bank_record` whose unique name is `my_record`

- **Note** - You can never assume the value of an uninitialized variable and the same is true of the individual fields of a struct. You should always initialize a struct after it has been declared

The individual fields of a struct can be accessed using the (dot) `.` notation

- To assign a value of 199.99 to `current_balance` field of the variable `my_record`, we use

```
my_record.current_balance = 199.99;
```

INITIALIZING STRUCTS

- It may be useful to define a function for **initializing** your structs

```
#include <iostream.h>
// Define the struct bank_record
struct bank_record
{
    int acc_number;
    int date_of_birth;
    float current_balance;
};
// prototype the initialize function
bank_record initialize();
main()
{
    bank_record john; // Declare an
                     // instance
    john = initialize(); // Initialize John
}

bank_record initialize()
{
    // Declare temporary local struct
    bank_record temp_rec;

    // Initialize all member variables
    temp_rec.acc_number = 0;
    temp_rec.date_of_birth = 0;
    temp_rec.current_balance = 0.0;

    // Pass struct back
    return temp_rec;
}
```

INITIALIZING STRUCTS

- ❑ Structs can also be initialized at the time of **declaration**, in a very similar way to arrays

To initialize an instance of a `bank_record` we would type

```
bank_record my_record =  
{  
    12345,  
    171271,  
    99.99  
};
```

STRUCTS AND FUNCTIONS

- ❑ Structs can be **passed to**, and **returned from**, functions in the same way as any other variable type, e.g.

```
bank_record find_record(int acc_num);
```

is a function that takes an account number and returns the necessary instance of `bank_record`

- ❑ Equality (`==`) and non-equality (`!=`) are **not defined** for structs. Although the programmer may define these operations (or define function that will compare them)
- ❑ **sizeof** operator on struct may return different size than sum of size of all struct members! Reasons behind are architecture dependent (some types are aligned to certain boundaries in memory).

ARRAYS OF STRUCTS

□ We have seen

- structs are a useful way of grouping together records of information
- arrays provide a way to store a collection of one particular type of object

□ By placing structs in arrays, we have a way of collecting together various records

To **declare** an array of `bank_record` structs, use

```
bank_record bank_file[10];
```

making `bank_file` an array of size 10 where each element in the array is a `bank_record`

□ How do we access each of the **individual** structs

To place a value into `current_balance` of the fifth record in the array

```
bank_file[4].current_balance = 99.99;
```

UNIONS

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

Usage:

```
union unique_name {member_list};
```

Example:

```
union WeightType
{
    long    wtInKgs;
    int     wtInPounds;
    float   wtInTons;
};
```

MORE ON UNIONS

- At run time, the memory space allocated to the variable `weight` does not include room for three distinct components.
- The assumption is that the program will never need a weight in kgs, a weight in pounds, and a weight in tons simultaneously while executing.
- The purpose of a union is to conserve memory by forcing several values to use the same memory space, one at a time.

Example:

```
weight.wtInTons = 4.83;
// weight in tons is no longer needed. Reuse the memory space.
weight.wtInPounds = 35;
// weight in pounds is no longer needed. Reuse the memory space.
weight.wtInKgs = 4000;
```

MORE ON STRUCTS AND UNIONS

- Due to historical reasons (C language definition), it's possible to have same name for struct (or union, enum) and nonstruct in same scope!
- Like here:

```
struct stats { int a; int b; };
int stats(int buff, int stats);
union one {int a; long b; long long c;};
long one(int a; union one b);
```

- In such case, plain name (`stats`) refer to function.
- To access structure, we must prefix it with `struct`.

```
main{
{
    struct stats my_stats;
    union one x;
    int y = stats(0,0);
    long z = one(0, x);
}
```

MORE ON STRUCTS AND UNIONS

- Type equivalence in case of structures is as follows:

```
struct one {int a};
```

```
struct two {int a};
```

```
one x;
```

```
two y,z;
```

```
y = x; // Error, different types!!!
```

```
y = z; // Correct, they have same type.
```

- Structures are also different from fundamental types

```
int I = y; // Wrong, y has type two, while I has type int
```

- Structs can be declared before they are defined much like functions.

```
struct bank_record;
```

```
void test();
```

```
struct bank_record { int a,b,c};
```

MORE ON TYPES

- There is a way how to define new name for type
- Keyword **typedef** will do a job.

```
typedef unsigned long ulong;
```

```
typedef int int32;
```

```
typedef short int16;
```

- Similar to way how reference variables works.
- New name is synonym for original one. May be freely mixed in program.
- Used to shorten type names (ulong is definitely shorter than unsigned long).