Structs

Last Time

- Today we will
  - Look at a new way of storing data called structs (short for structures)

Arrays and Data Types

- Useful for storing a collection of data elements of the **same** data type (float, int, string).

```
char myName[5];     //All elements chars
float salaries[NUM_EMP];  //All elements
  floats
char vowels[]={'A','E','I','O','U'};
```

- What about storing a collection of data elements of **different** data types?

Data with Different Data Types

- For example, what if we wanted to keep the following information on a particular employee:
  - employee id
  - SS#
  - number of children
  - salary
  - citizen
- The elements have different data types, so we can't conveniently use array. Instead we will use a **struct** (short for structure).

Structure Definition

To store this information:

- employee id
- SS#
- number of children
- salary
- citizen

We would begin by defining a structure :

```
struct employ
{
  int id
  int ssnum;
  int numchild;
  float salary;
  bool citizen;
};
```


Struct Terminology

For this struct:

```
struct employ
{
  int id
  int ssnum;
  int numchild;
  float salary;
  bool citizen;
};
```

- `employ` is the **identifier name** and a NEW data type.
- The individual components id, ssnum, etc. are called **members.**

Struct Declaration

- As with all data types, in order to use our new data type `employ` we must **allocate** storage space by **declaring** variables of this data type:

```
employ engineer, tech;
```

- This will allocate space for two variables called `engineer` and `tech` with the previously described members `id`, `ssnum`, etc.

Member Access Operator

- To access a struct member, we use the **member access operator** (period between struct variable name and member name).
- In the variable `engineer` of data type `employ` we can make the assignments:

```
engineer.id = 12345;
engineer.ssnum = 534334343;
engineer.numchild = 2;
engineer.salary = 45443.34;
engineer.citizen = true;
```

- How do we access the data in arrays?

Example One

- Write a C++ struct data type `realnum` that will have members `number`, `realpart`, and `intpart`.
- Declare a variable `numinfo` of that type.
- Place the value 3.14159 in the field `number`.

Example One Solution

```
struct realnum
{
   float number;
   int realpart;
   int intpart;
};

realnum numinfo;

numinfo.number=3.14159;
```

Define struct `realnum`

Declare variable `numinfo`

Assign member `number`

Structs as function arguments

- Structs can be passed to functions by reference or value in the same manner that other data types have been passed.
- Generally, passing structs by reference is preferred since passing by value requires a local copy of the struct be created within the function's variables.

Example Two

- Write a C++ function `split` that accepts a variable of type `realnum`.
- Assign the integer part of the number to the member variable `intpart` and the real part of the number to the member variable `realpart`.

Example Two Solution

- Function prototype:

```
void split(realnum &);
```

- Function call:

```
split (numinfo);
```

- Function definition:

```
void split(realnum & numberinfo)
{
// Use numberinfo.number,numberinfo.intpart,
// and numberinfo.realpart.
}
```


Example Three

Consider the following struct data type:

```
struct info
{
  int num;
  int divisors[10];
  int howmany;
};
```

Write a C++ function `compute` that accepts a variable of type `info` and returns all the divisors greater than 1 of the variable `num` in the array `divisors` and the number of divisors in the variable `howmany`.

Summary

- In today's lecture we covered
  - structures