CSCI 240 Lecture Notes - Part 12

A Last Clarification on Arrays

We have noted that arrays are passed differently than simple variables. In fact, that's a lie.

In C++, **everything** is passed by copy. For example:

0. When we code

```
fn(i);
```

where i is a simple variable, we are passing a **copy of the value of i.**

1. When we code

```
fnA(ar);
```

where ar is the name of an array, we are really passing a **copy of the address of the array**.

2. When we code

```
fnB(&i);
```

where i is the name of a simple variable, we are again passing a **copy of the addressof i.**

3. When we code

```
fnC(x);
```

where *fnC()* defines its argument as a reference argument, C++ is actually supplying **a copy of the address of x** when the function is called.

But in all cases, **copies** are passed.

Passing *reference* variables is a little different. While it is true that a copy of the address is passed, the compiler generates code so that if you assign something to that reference argument, the value-referred-to gets changed (not the address in the reference argument).

Again: any assignment to the passed-in reference argument alters the data at the address passed. So although the function gets a copy of the address of the data, the function cannot change the value of that copy (of the address).

Structures

A **structure** is a custom data type defined by the programmer.

It can hold a collection of variables of **different data types** (unlike arrays, in which all the data elements are the same type).

The data items in a structure are usually related (different kinds of information about a *person*, or about a *part*, or about an *account*, etc...).

Each data item in a structure is called a **member**. Sometimes these members are also called **fields**.

There are several minor variations in structure declarations. We will use the following:

```
struct new-type-name
  {
  data-type data-name1;
  data-type data-name2;
  ...
  };
```

Write this before your *main()* function and before any function prototypes that use this new type. Then it will be "known" to the compiler for the rest of the compilation. If you declare a struct inside *main()*, for example, it will only be "known" inside *main()* - and thus unusable in any other function.

Then to *declare* a variable of this new type, declare in any function that needs one:

```
new-type-name aVar;
```

A specific example:

```
struct Acct
  {
  char acctNum[10];
  char acctName[40];
  char acctType;
  double acctBal;
  };


int main()
  {
  Acct myAcct;              //declare a single variable of this new type.
  ...
  }
```

Note that the *struct definition* does not create any variable at all. It is just a template or a pattern. It takes up no memory and has no address. You still must *declare* one or more variables of this new type before you can use it.

Note also that after the above declarations, although we have a variable, it has no defined values in any of its members.

Again, the *struct* definition can be placed before the *main()* function so it is visible to all functions in the program. However, *instances* of variables of this type should normally be declared in the appropriate function or passed as arguments when necessary.

To reference a member of a structure, there is a "dot notation"

```
myAcct.acctBal = 1000.00;
```

sets the *acctBal* member in the structure *myAcct* to the value 1000.00.

```
strcpy(myAcct.acctName, "Jane Doe");
```

sets the *acctName* member of *myAcct* to "Jane Doe".

Note that we have an array of char (i.e. a string) as a structure member. Because it is a string, we have to use appropriate means to manipulate it.

We could not do this:

```
myAcct.acctName = "Jane Doe";           // NO NO
```

We could do this (but why bother?):

```
myAcct.acctName[0] = 'J';
myAcct.acctName[1] = 'a';
myAcct.acctName[2] = 'n';
...
myAcct.acctName[8] = '\0';
```

It is allowed to do compile time initialization (similar to other variables) as follows:

```
Acct anAcct = {"123456789", "Jane Doe", 'S', 1000.00};
```

Note that the data type of each member is clear.

The order must match the structure definition.

Omitted members (only at the end, if any) are set to 0's (for strings, null

However, this kind of initialization is rarely very useful, except for small test programs.

We can use *sizeof* on whole structures:

```
sizeof( Acct );            //the type
sizeof myAcct;                 //a variable of that type
```

Both evaluate to the number of bytes occupied by the structure.

Arrays of structs

It is perfectly acceptable and sensible to create an *array of structs*:

```
Acct ar[20]          //an array that can hold 20 Acct structs
```

Refer to them as:

```
//for the whole struct

ar[i]


//for the acctBal member of the struct in ar[i]

ar[i].acctBal
```

To pass an array of structs to a function, you can use the array notation.

```
void fn( Acct empAr[] )
{
empAr[3].acctBal = 300.00;         //for example
}
```

The calling statement would be:

```
fn( array_of_structs_name );
```

Returning Structures from Functions

It is allowed to create a structure as a local variable in a function and then return it to the calling program:

Declare a function:

```
Acct fn();
```

then in main:

```
Acct rec;

rec = fn();             // rec has valid values created by fn()
```

and the function:

```
Acct fn()
{
Acct r = {"123", "John Doe", 'C', 200.01};

return r;
}
```

Passing Structures to Functions

We can pass structures by copy (value) or by reference.

We usually pass by reference because:

- structures tend to be large, and making the copy wastes time and memory
- usually we will want to *change* some member or members of the structure

But if a structure is small and/or need not be modified, we can pass by value.

Suppose we want to write a function, *incrBal()*, to increment the account balance by some percentage (monthly interest or some such).

Call-by-Reference

```
void incrBal( Acct &anAcct, double percentage )
{
anAcct.acctBal = anAcct.AcctBal * (1.0 + percent);
}
```

The call would look like this:

```
incrBal(oneAcct, 0.06);
```

In the case of this particular task, since there is just one thing in the structure being altered, we could take another approach: just pass a reference to the *acctBal* member and write a new version of the function as:

```
void incrBal( double &bal, double percent )
{
bal = bal * (1.0 + percent);
}
```

and call it like:

```
incrBal(oneAcct.acctBal, 0.06);
```