STRUCTURES : defining, accessing, unions, fields, linked lists

---

Structures, also known as "records", provide the means to group related data of different types together in a convenient package. Structures also represent the basic building blocks of C++ classes.

DEFINING STRUCTURES

A structure definition consists of the 'struct' keyword and the structure name followed by a list of variable declarations:

```
struct tag

   variable declarations
optional_struct_declaration;
```

The following is an example of a structure definition:

```
struct employee

   char *lastName;
   char *firstName;
   char *streetAddress;
   char *city;
   char *state;
   long zipCode;
   int   employeeID;
   int   age;
;
```

The above definition only defines a structure of type 'employee'. To declare a variable which is a structure of type 'employee', the following declaration is used:

```
struct employee person; // declares and allocates memory for structure
```

Many times, pointers to structures are used.

```
struct employee *personPtr; // declares pointer only
```

If your working with pointers to structures, you must follow your pointer variable declaration with code to allocate memory for the structure. If your using straight C, you'd use 'malloc' from the ANSI library to allocate the memory for the structure like this:

```
personPtr = (struct employee *)malloc( sizeof( struct employee ) );
```

With C++, you use the 'new' operator to allocate the memory as follows:

```
personPtr = new employee;
```

The C++ way is obviously a better way of allocating memory and should be used in place of the old 'malloc' command.

ACCESSING STRUCTURE VARIABLES

If you've declared a structure variable, you access the variable members using the '.' member operator like this:

```
person.employeeID = 10;
person.age = 35;
```

You can use structure members just like any other variable.

```
if( person.age > 65 )
  DoSendPinkSlip( person );
```

When using pointers to structures, you access the variable members using the '->' member operator in a similar manner.

```
personPtr->employeeID = 10;
personPtr->age = 35;
```

UNIONS

Unions allow you to store different data type in the same memory location (but not at the same time). They follow the same syntax as structures:

```
union optional_tag

  variable declarations
optional_union_declaration;
```

The following is an example of a union definition:

```
union myUnionDef

  int     integer;
  float real;
  char   *string;
myUnion;
```

The memory allocated for a union will be large enough to hold the largest item declared in the union definition. Unions might be a good idea for maximizing memory use, but generally, they are rarely used and should probably be avoided.

BIT FIELDS

Fields, like unions, help to maximize memory usage but in days where memory is cheap and code clarity is important, fields are used less and less. They are implementation-dependent and create cross platform compatibility problems. Here's an example of a field definition:

```
struct sysFlags

  unsigned int SysFlag1 : 1;
  unsigned int SysFlag2 : 1;
  unsigned int SystemID : 2;
  unsigned int          : 4;   // unused
  unsigned int SysFlag3 : 1;
;
```

LINKED LISTS

Linked lists are a series of structures which are usually dynamically allocated and linked together (by pointers) to maintain a bond. Consider the following structure definition:

```
struct student
```

```
    struct student *next;
    char               *name;
    int                id;
;
```

When defining a new student, you would "link" the student to the existing list of students by doing something like this:

```
struct student *gTopOfList;       // In this example, program keeps track
struct student *gEndOfList;       // of top and bottom of linked list.
struct student *freshman;

freshman = new student;

gEndOfList->next = freshman;     // Last student is now new freshman.
gEndOfList = freshman;             // Update global variable.
gEndOfList->next = nil;           // Identifies end of list.
```

Using this example, you could print the entire student roster with the following code fragment:

```
struct student *currentStudent;

currentStudent = gTopOfList;
do
   cout << currentStudent->name << endl;
 while( currentStudent->next != nil );
```

An improved linked list would include a "prev" pointer in addition to the "next" pointer. This allows you to travel up as well as down the linked list chain of structures. There are tree linked lists, circular linked lists, and quite possibly, an infinite number of types and combinations you could come up with.