CS 101: Computer Programming and Utilization

Jul-Nov 2017

Umesh Bellur

Structures & Unions (Chapter 17++)

Recall the Student marks query program

- We used two arrays – one holding roll number and the other marks of the students.

- The programmer had to ensure that the two were tightly synchronized – i.e., the student whose roll number was at index **x** in the first array had to have the marks at index **x** in the second array.

- Often, we wish to store related information about ***entities*** in the problem domain.
  - Example: Customer Address, Customer ID, Customer Phone number of many customers.

- Sometimes we may also want to bind together ***functions*** that only operate on a specific entity's attributes.
  - Example: changeCustomerAddress, addPhoneNumber

Outline

- **Structure**
  - Basic facility provided in C++ to conveniently gather together information associated with an entity.
  - Inherited from the C language
- **Member functions**
  - New feature introduced in C++

Structures: Key Ideas

- Structure = collection of variables (aka attributes or members) denoting a new user defined type.
- Structure = ***super variable***, denotes the memory used for all members
- Each structure has a **name**, the name refers to the super variable, i.e. entire collection and denotes the type bring introduced.
- Each structure has a set of **Attributes**: each of a previously defined type.

Why `structs`?

- Open-ended user definable types

- Self-referential data structures can allow us to make useful structures
  - Lists, trees, etc.

Structure Types

- You can define a structure type for each entity that you want to represent on the computer
  - Example: To represent books, you can define a Book structure type, to represent students, you define a student data type and so on.

- When you define a structure type, you must say what variables each structure of that type will contain
  - A book type has a ISBN number, Publisher, Publish Date
  - A student has a name, roll number, address, current CPI, etc.

Defining a structure type

General form

```
struct structure-type{
   member1-type member1-name;
   member2-type member2-name;
     ...
};              // Don't forget the semicolon!
```

Example

```
struct Book{
   char title[50];
   double price;
};
```

Book is a user-defined data type, just as int, char, double are primitive data types

Structure-type name and member names can be any identifiers

Self Referential….

Yes! This is legal!

```
struct item {
  char *s;
  struct item *next;
};
```

- I.e., an **`item`** can point to another **`item`**
- … which can point to another **`item`**
- … which can point to yet another **`item`**
- … etc.

Thereby forming a *list* of **`items`**

Creating Structures of A Type Defined Earlier

To create a structure variable of structure type Book, just write:

```
Book p, q;
```

This creates two structures: p, q of type Book.

Each created structure has all members defined in structure type definition.

Accessing members of a structure

```
p.price = 399;    // stores 399 into p.price.
cout << p.title;  // prints the name of the book p
```

Initializing structures

```
Book b = { "On Education" , 399};
Book b2 = {"c++ made easy"}; // correct
Book b3 = {456}; // Error
```

Stores "On Education" in b.title (null terminated as usual) and 399 into b.price

A value must be given for initializing each leading member

Just like with all variables, you can make a structure unmodifiable by adding the keyword const:

```
const Book c = { "The Outsider" , 250};
```

One Structure Can Contain Another

```
struct Point{
   double x,y;
};
struct Circle{
   Point center;      // contains Point
   double radius;
};
Circle c1;
c1.radius = 10;
c1.center = {15, 20};
// sets the x  and y members of center member of d
```

Assignment

One structure can be ***assigned*** to another

- All members of right hand side copied into corresponding members on the left
- Structure name stands for entire collection unlike array name which stands for address
- A structure can be thought of as a (super) variable

```
book b = { "On Education" , 399};
book c;
c = b;        // all members copied
cout << c.price << endl;  // will print 399
```

Arrays of Structures

```
Circle c[10];
Book library[100];
```

Creates arrays c, library which have elements of type Circle and Book

```
cin >> c[0].center.x;
```

Reads a value into the x coordinate of center of 0th Circle in array c

```
cout << library[5].title[3];
```

Prints 3rd character of the title of the 5th book in array library

Structures and Pointers

```
Book  b1;
Book* b2 = &b1;

b2 -> price = 140.99;
```

OR

```
(*b2).price = 140.99;
```

Because '.' operator has higher precedence than unary '*'

Structures and Functions

- Structures can be passed to functions by value (members are copied), or by reference

- Structures can also be returned. This will cause members to be copied back

Parameter Passing by Value

```
struct Point{double x, y;};
Point midpoint(Point a,  Point b){
  Point mp;
  mp.x = (a.x + b.x)/2;
  mp.y = (a.y + b.y)/2;
  return mp;
}

int main(){
  Point p={10,20},  q={50,60};
  Point r = midpoint(p,q);
  cout << r.x << endl;

  cout << midpoint(p,q).x  << endl;
}
```

Note that the return value is copied over to r

Note that the temporary structure is used as RHS

Parameter Passing by Reference

```
struct Point{double x, y;};
Point midpoint(const Point &a, const Point &b){
   Point mp;
   mp.x = (a.x + b.x)/2;
   mp.y = (a.y + b.y)/2;
   return mp;
}
int main(){
   Point p={10,20},   q={50,60};
   Point r = midpoint(p,q);
   cout << r.x << endl;
}
```

A Structure to Represent 3 Dimensional Vectors

- Suppose you are writing a program involving velocities and accelerations of particles which move in 3 dimensional space
- These quantities will have a component each for the x, y, z directions
- Natural to represent using a structure with members x, y, z

```
struct Vec{ double x, y, z; };
```

Using Struct Vec

Vectors can be added or multiplied by a scalar. We might also need the length of a vector.

```
Vec sum(const Vec &a, const Vec &b){
   Vec v;
   v.x = a.x + b.x;
   v.y = a.y + b.y;
   v.z = a.z + b.z;
   return v;
}
Vec scale(const Vec &a, double f){
   Vec v;
   v.x = a.x * f; v.y = a.y * f; v.z = a.z * f;
   return v;
}
double length(const Vec &v){
   return sqrt(v.x*v.x + v.y*v.y + v.z*v.z);
}
```

Motion Under Uniform Acceleration

If a particle has an initial velocity u and moves under uniform acceleration a, then in time t it has a displacement $s = ut + at^2/2$, where u, a, s are vectors

To find the distance covered, we must take the length of the vector s

```
int main(){
   Vec  u, a, s;   // velocity,
                   // acceleration,
                   // displacement
   double t;    // time
   cin >> u.x >> u.y >> u.z >>
          a.x >> a.y >> a.z >> t;
   s = sum(scale(u,t), scale(a, t*t/2));
   cout << length(s) << endl;
}
```

Member functions

- Rather than creating functions that operate on structs, we sometimes find it useful to BIND these functions to the structure
  - When the function is relevant only to that collection of attributes.

- In C++, you can make the functions ***<u>a part of the struct definition itself</u>***. Such functions are called **member functions**.

- By collecting together relevant functions into the definition of the struct, the code becomes better organized

Structures with Member Functions

```
struct Vec{
 double x, y, z;
 double length(){
  return sqrt(x*x + y*y + z*z);
 }
};

int main(){
 Vec  v={1,2,2};
 cout << v.length() << endl;
}
```

Length is the member function

V is the receiver of the call to length.

References to member variables of the receiver.

The Complete Definition of Vec

```
struct Vec{
   double x, y, z;
   double length(){
      return sqrt(x*x + y*y + z*z);
   }
   Vec sum(Vec b){
      Vec v;
      v.x = x+b.x; v.y=y+b.y; v.z=z+b.z;
      return v;
   }
   Vec scale(double f){
      Vec v;
      v.x = x*f; v.y = y*f; v.z = z*f;
      return v;
   }
}
```

Notice it takes only one Vec as an argument

Notice it takes only the scaling factor as an argument

Main Program

```
int main(){
 Vec u, a, s;
 double t;
 cin >> u.x >> u.y >> u.z >> a.x >> a.y
      >> a.z >> t;
 Vec ut = u.scale(t);
 Vec at2by2 = a.scale(t*t/2);
 s = ut.sum(at2by2);
 cout << s.length() << endl;

// green statements equivalent to:
 cout << u.scale(t).sum(a.scale(t*t/2)).length()
       << endl;
}
```

One More Example: Taxi Dispatch

- Problem statement: Clients arrive and have to be assigned to (earliest waiting) taxies
- An important part of the solution was a blackboard on which we wrote down the ids of the waiting taxies
- How would we implement this using structs?

  What structures should we create???

One More Example: Taxi Dispatch

- Customers are assigned taxis immediately if available

  Information about customers never needs to be stored

- Each taxi is associated with just one piece of information: id . We can just use an int to store the id

- The blackboard however is associated with a lot of information: array of ids of waiting taxis, front/last indices into the array

  So we should create a struct to represent the blackboard

Representing the Blackboard

```
const int N=100;
struct Queue{
 int elements[N],
     nwaiting,
      front;

 bool insert(int v){
  …
  }

 bool remove(int &v){
  …
 }
};
```

- N = max no. of waiting taxis
- We call the struct a Queue rather than blackboard to reflect its function
- nwaiting = no. of taxis currently waiting
- front = index
- Elements[N] holds the IDs of the waiting taxis.
- Two operations on Queue: inserting elements and removing elements.

  These become member functions

Member Function Insert

```
struct Queue{
  …
  bool insert(int v){
      if(nWaiting >= N) return false;
      elements[(front + nWaiting)%N] = v;
      nWaiting++;
      return true;
  }
};
```

- A value can be inserted only if the queue has space
- The value must be inserted into the next empty index in the queue
- The number of waiting elements in the queue is updated
- Return value indicates whether operation was successful

Main Program

```
int main(){
 Queue q;
 q.front = q.nWaiting = 0;
 while(true){
  char c; cin >> c;
  if(c ==  ‘d’ ){
    int driver;
    cin >> driver;
    if(!q.insert(driver))
         cout << “Q is full\n” ;
  }
   else if(c ==  ‘c’ ){
    int driver;
    if(!q.remove(driver))
         cout << “No taxi available.\n” ;
    else cout << “Assigning <<driver<< endl;
  }
}
```

A note on precedence of Operators . -> []

```
struct triangle tr, *trp=&tr;

tr.pkt1.x
trp->pt1.x
(tr.pt1).x
(trp->pt1).x    // Are all equivalent

++trp->pt1.x;   // will increment x by 1
```

Example: Arrays of structures

- Use of two parallel arrays

```
char *keyword[NKEYS]; /* keywords */
int keycount[NKEYS]; /* counters of keywords */
```

- use of structures

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

OR

```
struct key {
char *word;
int count;
};
struct key keytab[NKEYS];
```

Unions

- A **`union`** is like a **`struct`**, but only one of its members is stored, not all
  - I.e., a single variable may hold different types at different times
  - Storage is enough to hold largest member
- E.g.,

```
union {
  int ival;
  float fval;
  char *sval;
} u;
```

Unions (continued)

- It is *programmer's responsibility* to keep track of which type is stored in a **`union`** at any given time!
- E.g.,

```
struct taggedItem {
  enum {iType, fType, cType} tag;
  union {
   int ival;
   float fval;
   char *sval;
  } u;
};
```

Members of **`struct`** are:–

`enum tag;`

`union u;`

Value of **`tag`** says which member of **`u`** to use

Union Types

- E.g.,

```
typedef union{
    bool wears_wig;
    char color[20];
} hair_t;
```

- Suppose we declare a variable

```
hair_t hair_data;
```

- `hair_data` contains either the `wears_wig` component or the `color` component but not both.

A Function Using a Union Structure

- Suppose we have a structure variable.

```
Struct hair_info_t {
    bool bald;
    hair_t h;
};
```

- We can use this structure to reference the correct component.

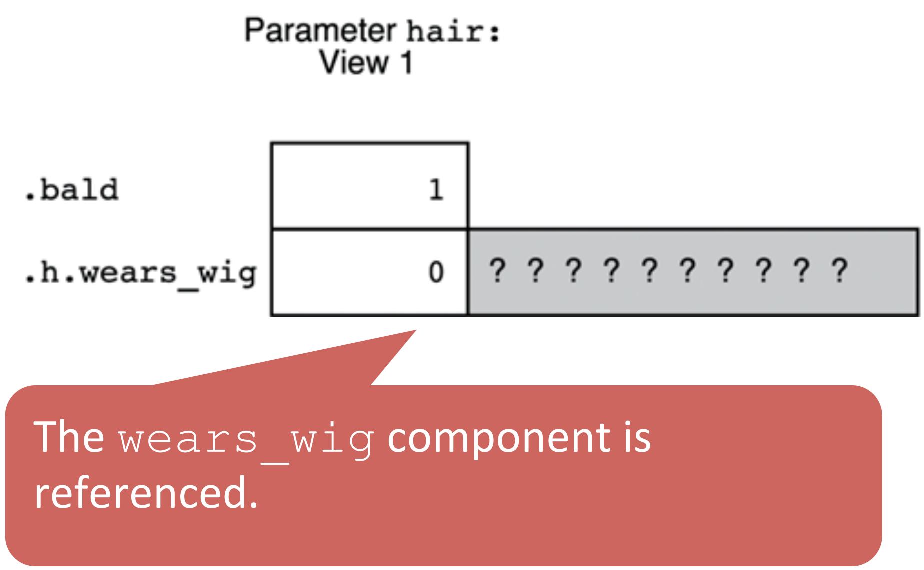
```
void
print_hair_info(hair_info_t hair) /* input - structure to display         */
{
      if (hair.bald) {
            printf("Subject is bald");
            if (hair.h.wears_wig)
                  printf(", but wears a wig.\n");
            else
                  printf(" and does not wear a wig.\n");
      } else {
            printf("Subject's hair color is %s.\n", hair.h.color);
      }
}
```

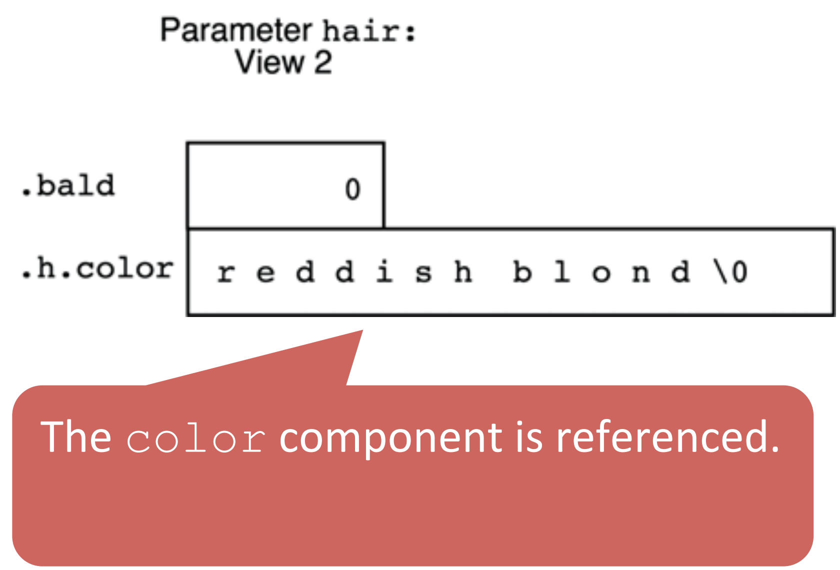
Use the `wears_wig` component.

Use the `color` component.

Two Interpretations of the Union Variable `hair_t`

- The memory content of `hair_t` depends on which component is referenced.
  - The memory allocated for `hair_t` is determined by the **largest** component in the union.





Unions (continued)

- **`unions`** are used much less frequently than **`structs`** — mostly
  - in the inner details of operating system
  - in device drivers
  - in embedded systems where you have to access registers defined by the hardware