STRUCTURES

CS 23200

Outline (Structures)

- Syntax and initialization
- Assignment
- Structures and functions
- Nesting
- Pointers to structures and arrays of structures
- Self-referential structures

Structures

- Structures group together related data
  - As close as C gets to objects

structure tag: can create structures of this type later

structure declaration: tells what goes in a "struct room"

```
struct room {
   char building[5];
   int number;
   int capacity;
};  /* remember the semicolon! */
```

structure declaration: declares a variable "aRoom" of type "struct room"

```
struct room aRoom;
```

Different Declaration Forms

```
struct room {
   char building[5];
   int number;
   int capacity;
};

struct room aRoom;
```

- This is the preferred form for structure declarations
  - Separates the template specification from the variable declaration
- Other forms are possible

Structure Declaration and Initialization

```
struct room {
   char building[5];
   int number;
   int capacity;
};
```

```
struct room someRoom;
/* members are uninitialized */

struct room thisRoom =
   {"KT", 225, 30};
/* what if number and capacity
change order in the structure? */

struct room sameRoom;
strncpy(sameRoom.building,
   "KT",
   sizeof(sameRoom.building));
sameRoom.number = 225;
sameRoom.capacity = 30;
```

The . accesses the members of the structure

Outline (Structures)

- □ Syntax and initialization
- □ Assignment
- □ Structures and functions
- □ Nesting
- □ Pointers to structures and arrays of structures
- □ Self-referential structures

Structure Assignment

```
struct room {
   char building[5];
   int number;
   int capacity;
};

struct room thisRoom = createRoom("KT", 225, 30);
struct room sameRoom;
sameRoom = thisRoom;
```

- □ Structure assignment copies each field
  - Contents of `building` are copied
  - `sameRoom.number = thisRoom.number;`
  - `sameRoom.capacity = thisRoom.capacity;`
- □ Comparing structures (`==` or `!=`) is not allowed

Outline (Structures)

- □ Syntax and initialization
- □ Assignment
- □ Structures and functions
- □ Nesting
- □ Pointers to structures and arrays of structures
- □ Self-referential structures

Structures and Functions

- Functions can return structures

```
struct room createRoom(const char* building,
  const int number, const int capacity);
```

- Functions can take structures as arguments

```
int getComfortableCapacity(struct room theRoom){
   return (int)(theRoom.capacity * 0.9);
}
```

- Can do both in the same function

```
struct room upOneFloor(struct room theRoom){
   struct room roomAbove = theRoom;
   roomAbove.number += 100;
   return roomAbove;
}
```

Outline (Structures)

- Syntax and initialization
- Assignment
- Structures and functions
- Nesting
- Pointers to structures and arrays of structures
- Self-referential structures

Nesting Structures

- Nesting: structures as members of other structures

```
struct room {
   char building[5];
   int number;
   int capacity;
};

struct course {
   char dept[5];
   int number;
   char *name;
   struct room theRoom;
   int enrollment;
};
```

```
struct course aCourse;
...
aCourse.name = "Intro to C/UNIX";
aCourse.theRoom = thisRoom;
aCourse.enrollment = 15;

if(aCourse.enrollment >
   aCourse.theRoom.capacity){
   /* print an error message */
...
```

Outline (Structures)

- Syntax and initialization
- Assignment
- Structures and functions
- Nesting
- Pointers to structures and arrays of structures
- Self-referential structures

An Example

```
struct bigStruct {
   char name[1000];
   int ids[1000];
};
void changeIt(struct bigStruct s){
   s.ids[0] = 17;
}
int main(){
   struct bigStruct s;
   ...
   changeIt(s);
```

- □ What is misleading about this code?
- □ Pass-by-value: s is not changed in main

An Example

```
struct bigStruct {
   char name[1000];
   int ids[1000];
};
struct bigStruct changeIt(struct bigStruct s){
   s.ids[0] = 17;
   return s;
}
int main(){
   struct bigStruct s;
   ...
   s = changeIt(s);
```

- □ This works, but it copies `bigStruct` twice
  - ◘ The local copy for the function (pass-by-value)
  - ◘ The assignment statement in `main`

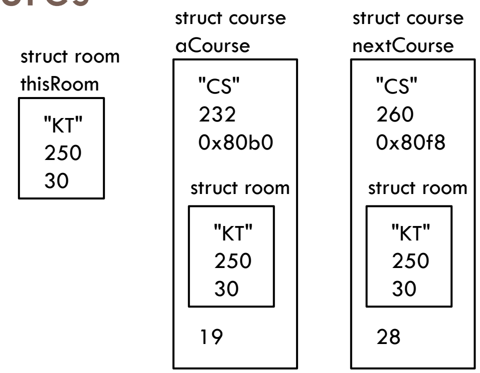
Pointers to Structures

```
void changeIt(struct bigStruct *ps){
   ps->ids[0] = 17;
}
int main(){
   struct bigStruct s;
   ...
   changeIt(&s);
```

- □ Avoids copying entire structure
- □ The arrow operator (`->`) works on pointers to structures
  - ◘ Accesses the structure members
  - ◘ `ps->whatever` is the same as `(*ps).whatever`
- □ Address-of operator (`&`) works on structures

Pointers to Structures

```
struct room {
   char building[5];
   int number;
   int capacity;
};
struct course {
   char dept[5];
   int number;
   char *name;
   struct room theRoom;
   int enrollment;
};
```

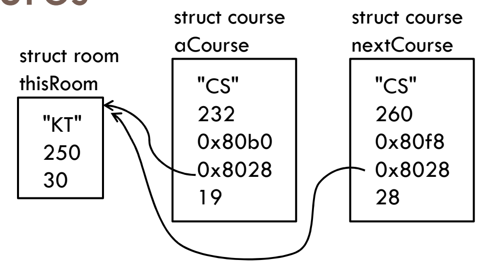


```
struct room thisRoom;
struct course aCourse;
struct course nextCourse;
...
aCourse.theRoom = thisRoom;
...
nextCourse.theRoom = thisRoom;
```

- □ Multiple copies of the data for `thisRoom`
  - ◘ Wastes space
  - ◘ Not robust: What if some function edits the capacity of `thisRoom`? (e.g., a temporary wall is put up)

Pointers to Structures

```
struct room {
   char building[5];
   int number;
   int capacity;
};
struct course {
   char dept[5];
   int number;
   char *name;
   struct room *pRoom;
   int enrollment;
};
struct room thisRoom;
struct course aCourse;
struct course nextCourse;
...
aCourse.pRoom = &thisRoom;
...
nextCourse.pRoom = &thisRoom;
```



struct room thisRoom
"KT"
250
30

struct course aCourse
"CS"
232
0x80b0
0x8028
19

struct course nextCourse
"CS"
260
0x80f8
0x8028
28

- □ Single copy of the data for `thisRoom`
  - If `thisRoom` is changed, all the courses' rooms reflect that change

Arrays of Structures

```
const int numRooms = 12387;
const int numCourses = 53289;
struct room theRooms[numRooms];
struct course theCourses[numCourses];
...
theRooms[roomID].capacity = 30;
...
theCourses[courseID].pRoom = &(theRooms[roomID]);
```

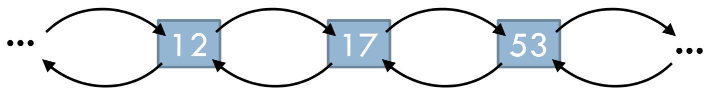
- □ Can declare arrays of structures
- □ Indexing is done as with other data types

Outline (Structures)

- □ Syntax and initialization
- □ Assignment
- □ Structures and functions
- □ Nesting
- □ Pointers to structures and arrays of structures
- □ Self-referential structures

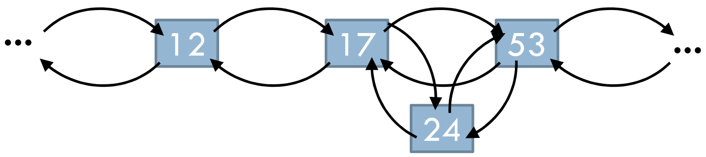
Self-referential Structures

- □ Example: doubly-linked list
  - An alternative to an array for storing a sequence of items
  - Array requires the elements be contiguous in memory
  - Linked list elements can be anywhere in memory, because each element points to the next and previous elements



Self-referential Structures

- Doubly-linked list insertion
  - Constant time (doesn't depend on the size of the list)
  - In an array, we would have to slide some elements down
    - Time to insert depends on the size of the list



Doubly Linked List

```
struct data {...};

~~struct listnode {~~
~~   struct data d;~~
~~   struct listnode prev;~~
~~   struct listnode next;~~
~~};~~
```

```
struct data {...};

struct listnode {
   struct data d;
   struct listnode *prev;
   struct listnode *next;
};
```

- A struct cannot contain a member of its own type
  - Because that member would have a member of its own type, which would have a member of its own type, …
- A struct can contain a member that is a **pointer** to its own type

```
struct data {...};

struct listnode {
   struct data d;
   struct listnode *prev;
   struct listnode *next;
};
```

- Write the following functions:
  - A function to insert a listnode after another listnode
    - Pointers to the two nodes are arguments to the function
    - Assume the node you insert after is not the last element in the list
  - A function to remove a listnode from the list
    - A pointer to the node is the only argument
    - Assume that the node to remove is neither the first nor the last element in the list

Insertion Example

```
struct data {...};

struct listnode {
   struct data d;
   struct listnode *prev;
   struct listnode *next;
};

void insert(struct listnode *pToInsert,
            struct listnode *pInsAfter){
   struct listnode *pInsBefore = pInsAfter->next;
   pToInsert->next = pInsBefore;
   pToInsert->prev = pInsAfter;
   pInsAfter->next = pToInsert;
   pInsBefore->prev = pToInsert;
}
```

Removal Example

```
struct data {...};

struct listnode {
   struct data d;
   struct listnode *prev;
   struct listnode *next;
};

void remove(struct listnode *pToRemove){
   pToRemove->next->prev = pToRemove->prev;
   pToRemove->prev->next = pToRemove->next;
}
```

Resources from CS50

- Singly-linked lists
  - https://www.youtube.com/watch?v=ZoG2hOIoTnA
- Doubly-linked lists
  - https://www.youtube.com/watch?v=HmAEzp1taIE

Summary

- Structures capture the natural hierarchy in real-world data
- Initialization: assign each member variable
- Can pass lots of information in and out of functions without using lots of parameters
  - Passing pointers to structures can avoid unnecessary copying
- Can use assignment (`=`), address of (`&`), and member (`.` or `->`) operators
- Cannot compare using `==` or `!=`
- Structures can be nested (structures inside other structures) and they can be self-referential (using pointers)

Big Picture

- ☑ Writing programs on *nix computers
  - ☑ File system commands
  - ☑ Editing, compiling, running, and debugging programs
- □ C Language
  - ☑ Familiar aspects of C (variables, operators, basic I/O, control flow, functions)
  - ☑ Basics of pointers and structures
  - Advanced pointers, memory management
  - More structures and related constructs
  - Multi-file programs
  - …