Structs, Unions, and Enums

CS449 Fall 2015
Data Type Review

• We learned...
  – Primitive data types: char, int, long, float, double...
  – Derived data types: pointers, arrays, pointers of arrays, arrays of pointers

• Today we will learn two new derived types
  – Structs
  – Unions

• And one “syntactic sugar” data type
  – Enums
Structs

• Struct: A derived type for a collection of related variables under one name
  – Much like classes in Java with no methods
  – Members can be primitives or derived types

• Useful for...
  – Readability of code through conceptual grouping
  – File I/O of fixed length records into structs
  – Designing recursive data structures (e.g. linked lists, trees, graphs) using pointers
#include <stdio.h>
struct Point{
    int x;
    int y;
};
void print_point(const struct Point *ppnt) {
    printf("pnt=(%d, %d)\n", ppnt->x, ppnt->y);
}
int main (int argc, char *argv[])
{
    struct Point pnt;
    pnt.x = 10;
    pnt.y = 20;
    print_point(&pnt);
    return 0;
}
Struct Type Declaration

```c
#include <stdio.h>

struct Point{
    int x;
    int y;
};

void print_point(const struct Point *ppnt) {
    printf("pnt=(%d, %d)\n", ppnt->x, ppnt->y);
}

int main (int argc, char *argv[])
{
    struct Point pnt;
    pnt.x = 10;
    pnt.y = 20;
    print_point(&pnt);
    return 0;
}
```

- This declares a type “struct Point”
  - Similar to a class declaration in Java
- The struct has two members “x” and “y”
- Note: no new variables (storage locations) have been declared yet
  - “x” and “y” are not storage locations
#include <stdio.h>
struct Point{
  int x;
  int y;
};
void print_point(const struct Point *ppnt) {
  printf("pnt=(%d, %d)\n", ppnt->x, ppnt->y);
}
int main (int argc, char *argv[])
{
  struct Point pnt;
  pnt.x = 10;
  pnt.y = 20;
  print_point(&pnt);
  return 0;
}
# Operations on Structs

```c
#include <stdio.h>

struct Point{
    int x;
    int y;
};

void print_point(const struct Point *ppnt) {
    printf("pnt=(%d, %d)\n", ppnt->x, ppnt->y);
}

int main (int argc, char *argv[])
{
    struct Point pnt;
    pnt.x = 10;
    pnt.y = 20;
    print_point(&pnt);
    return 0;
}
```

- Four valid operations on structs
  - The `sizeof` operator
  - The reference (`&`) operator
  - Accessing member variables
    - `"."` (dot) operator (When accessed through struct var)
    - `"->"` (arrow) operator (When accessed through pointer to struct var)
  - The assignment (`=`) operator (e.g. `struct Point x, y; x = y;`)

- Recall: arrays were allowed `sizeof`, `&`, and `[]` operators but did not allow assign

- Passing structs to functions
  - By pointer (copy just the pointer)
  - By value (copy the entire struct)
# Struct Initialization

```
#include <stdio.h>
struct Point{
    int x;
    int y;
};
void print_point(const struct Point *ppnt) {
    printf("pnt=(%d, %d)\n", ppnt->x, ppnt->y);
}
int main (int argc, char *argv[])
{
    struct Point pnt;
    pnt.x = 10;
    pnt.y = 20;
    print_point(&pnt);
    return 0;
}
```

- Good habit to initialize all structs just like any variable
- Three ways to initialize
  - Initialize members one by one
  - Through assigning to another struct
  - Through struct initializer (e.g. `struct Point pnt = {10, 20};`)
- Struct initializers can only be used at variable declaration time, just like array initializers
Recursive Member Definitions

• Structs cannot have members of own type
  
  ```c
  struct A {
    int x;
    struct A y;
  };
  ```
  ... is Illegal (Think of what the size of struct A would be.)

• Structs can have members of pointers to own type
  
  ```c
  struct A {
    int x;
    struct A* y;
  };
  ```
  ... is legal (Now think of what the size of struct A would be.)
Recursive Data Structures

- **Linked List:**
  ```c
  struct Node {
    int data;
    struct Node *next;
  };
  ```

- **Binary Tree:**
  ```c
  struct Node {
    int data;
    struct Node *left;
    struct Node *right;
  };
  ```
Mysterious Sizeof Struct Example

```c
#include <stdio.h>
int main (int argc, char *argv[]) {
  struct {
    char x;
    long long y;
  } A;
  printf("sizeof(A)=%d\n", sizeof(A));
  return 0;
}
• Why 16? Why not 9 (1 + 8)?
• Because the address of “y” needs to be aligned
```
Inefficiency of Word-Spanning Accesses

• Word: (largest) unit of data that can be accessed in a single memory operation
  – What it means to be a “32-bit” or “64-bit” system
  – Usually, size of word == register size
    • Efficient to load/store register in a single operation

• Problem: what if an access spans multiple words (lands on the boundary between two words)?
  – Would result in two memory accesses
    • Imagine patching together a value from two accesses
    • Imagine accesses to two different cache lines, pages etc
Alignment and Word Accesses

• n-byte aligned: address is n-byte aligned
  – if it is a multiple of n

• aligned (access): access is aligned
  – if address is n-byte aligned
  – if datum is n bytes long (where n is a power of 2)
  – then, aligned accesses never span words,
    if n <= word size
Alignment and Word Accesses

• aligned (primitive pointer): pointer $p$ is aligned
  – if $p$ points to a base type of $n$ bytes
  – if $p$ always points to an $n$-byte aligned address (even after performing pointer arithmetic on $p$)
  – then, accesses to $p$ must be all aligned accesses

• aligned (aggregate pointer): pointer $p$ is aligned
  – if pointer to each primitive member is aligned (even after performing pointer arithmetic on $p$)
  – For array: only requires first element to be aligned
  – For struct: more complicated (members differ in size)
Padding to Enforce Struct Alignment

- Compiler inserts padding to prevent misaligned accesses (Even when A is used in an array, hence 2\textsuperscript{nd} case)
- Different compilers may produce different padding ➔ Must be careful when writing/reading file using struct

```
struct {
  char x;
  long long y;
} A;

struct {
  long long y;
  char x;
} A;
```
Unions

• Union: A derived type for a variable that can store values of different data types
  – Syntax is exactly the same as structs
  – Members can be primitive or derived types
  – Each member begins at the same memory location (Members share space)
  – Update of member overwrites shared space
  – Only members last written to can be read

• Useful for...
  – Declaring storage space used for multiple purposes (e.g. storing a string, int, and float at different times)
  – Saves storage space
Union Example

```c
#include <stdio.h>
#include <string.h>
union Number {
    int num;
    char str[100];
};
int main (int argc, char *argv[])
{
    union Number number;
    number.num = 5;
    printf("%d\n", number.num);
    strcpy(number.str, "Five");
    printf("%s\n", number.str);
    printf("%d\n", number.num);
    return 0;
}
```
Size of union is at least as large as the largest member.
Enums

- Enumeration: data type consisting of a set of named values called enumerators. E.g.:
  - enum Suit {Spades, Diamonds, Clubs, Hearts} suit;
- In C, an enum is an alias for an integer type (size depends on compiler)
- Enumerators are aliases for integer constants
- Above variable declaration equivalent to:
  ```
  int suit;
  const int Spades = 0;
  const int Diamonds = 1;
  const int Clubs = 2;
  const int Hearts = 3;
  ```
- Can assign integer values to enumerators
  - enum Suit {Spades = 1, Diamonds, Clubs, Hearts}
  - enum Suit {Spades = 1, Diamonds = 2, Clubs = 4, Hearts = 8}
# Enum Example

```c
#include <stdio.h>
int main (int argc, char *argv[])
{
    enum Suit {Spades=1, Diamonds, Clubs, Hearts};
    enum Suit suit;
    for(suit = Spades; suit <= Hearts; ++suit) {
        printf("suit=%d\n", suit);
    }
    printf("Hearts=%d\n", Hearts);
    return 0;
}
```

```
>> ./a.out
suit=1
suit=2
suit=3
suit=4
Hearts=4
```
#include <stdio.h>
int main (int argc, char *argv[]) 
{
    enum Suit {Spades=1, Diamonds, Clubs, Hearts};
    enum Suit suit;
    for(suit = Spades; suit <= Hearts; ++suit) {
        printf("suit=%d\n", suit);
    }
    printf("Hearts=%d\n", Hearts);
    return 0;
}

- Since enum is really an integer, any arithmetic operation is permitted
- However, only comparison and increment/decrement operators make sense semantically
Operations on Enumerators

```c
#include <stdio.h>

int main (int argc, char *argv[])
{
    enum Suit {Spades=1, Diamonds, Clubs, Hearts};
    enum Suit suit;
    for(suit = Spades; suit <= Hearts; ++suit) {
        printf("suit=%d\n", suit);
    }
    printf("Hearts=%d\n", Hearts);
    return 0;
}
```

- Since an enumerator is really a integer constant, any arithmetic operation or assignment to integer variables
- However, only meant to be used in relation to the original enum type