C Structures

CS 2060

Prof. Jonathan Ventura

C Structures: `struct`

- C enables us to aggregate diverse data types into a *structure*:

```
struct Employee {
    char first_name[20];
    char last_name[20];
    unsigned int age;
    char gender;
    double hourly_salary;
};
```

- A `struct` is defined outside of any function.
- Note the construction with `struct`, the name, brackets, and semi-colon.

C Structures: struct

- To define a struct variable:

```
int main() {
    struct Employee employee1;
    strcpy( employee1.first_name, "Jane" );
    strcpy( employee1.last_name, "Doe" );
    employee1.age = 22;
    employee1.gender = 'f';
    employee1.hourly_salary = 20;
}
```

- Member variables are accessed using the . operator.

Relationship to Java/C++ classes

- A `struct` is similar to a class as in Java, C++, Python, C#, etc.
- However, a C `struct` cannot have member functions (methods) – it can only have member variables.
- It is merely a convenient way to organize a block of memory.

struct example

```
struct Message {
    char *sender;          // sender IP address
    char *recipient;       // recipient IP address
    size_t data_size;      // number of bytes
    char *data;            // data
    int timeout;           // network timeout
};
int sendMessage( const struct Message *msg );
int recvMessage( struct Message *msg );
```

Initializing a struct

- It is possible to initialize a struct with an initializer array:

```
struct DictionaryEntry {
    char *word;
    char *definition;
    int scrabble_score;
};
DictionaryEntry entry1 = { "apple", "a fruit", 9 };
```

Using typedef

- It is convenient to use typedef to make the struct into a named data type:

```
typedef struct {
    float x, y;
} Point2f;

int main() {
    Point2f pt;
    pt.x = 10;
    pt.y = 20;
}
```

- With the typedef we don't need to write struct before the name when defining variables.

struct arrays and pointers

- We can make arrays and pointers of structs:

```
Point2f points[10];
Point2f *ptr = points;
ptr->x = 10;
ptr->y = 20;
ptr++;
```

- Note the use of -> to directly access member variables with a struct pointer.

```
ptr->x = 10;
(*ptr).x = 10; // equivalent
```

sizeof

- `sizeof` tells us the size of the entire structure:

```
Point2f points[10];
memset( points, sizeof(Point2f)*10, 0 );
```

- What does `sizeof(Point2f)` return?
- What does this code do?

Passing a `struct` to a function

- We can pass a `struct` by value or by reference:

```
void sendMessageByValue( Message msg );
void sendMessageByReference( Message *msg );
```

- Why is pass-by-reference (using a pointer) preferred?

Passing a `struct` to a function

- We can pass a `struct` by value or by reference:

```
void sendMessageByValue( Message msg );
void sendMessageByReference( Message *msg );
```

- Pass-by-value will make a copy of the structure, which might be a large block of memory.
- Pass-by-reference (using a pointer) will not make a copy, so it is faster and more memory-efficient.

Passing a struct to a function

- structs are useful to package data together and reduce arguments to functions:

```
void sendMessageLong(
    char *sender, char *recipient,
    size_t data_size, char *data,
    int timeout );
void sendMessageShort( Message *msg );
```

- Using a struct, the data stays together and the code is less messy.

struct arrays

- struct arrays are also useful to keep data together:

```
char * words[10];
char * definitions[10];
int scrabble_scores[10];
// OR:
struct DictionaryEntry entries[10];
```

- Again, with a struct, the data stays together and the code is less messy.

struct example

```
typedef struct {
    char name[10];
    int score;
} Player;

void incrementPlayerScore( Player *player, int score )
{
    player->score += score;
}

void printScoreBoard( int nplayers, Player players[] )
{
    for ( int i = 0; i < nplayers; i++ ) {
        printf("%15s",players[i].name);
    }
    puts("");
    for ( int i = 0; i < nplayers; i++ ) {
        printf("%15d",players[i].score);
    }
}
```

struct example

```
int main()
{
    Player players[4];
    puts("Enter player names:");
    for ( int i = 0; i < 4; i++ ){
        scanf( "%10s", players[i].name );
    }

    int round = 1;
    int next_player = 0;
    while ( 1 ) {
        printf("*** ROUND %d ***\n",round);
        printf("Enter score for player %s:\n",players[next_player].name);
        int score;
        if ( scanf("%d",&score) != 1 ) break;
        incrementPlayerScore(&players[next_player],score);
        next_player = (next_player+1)%4;
        printScoreBoard(4,players);
    }
}
```

Self-referencing structures

- A `struct` can have a member variable whose type is a pointer to the `struct` itself:

```
struct Employee {
    char firstName[20];
    char lastName[20];
    struct Employee *manager;
};
```

- Here the `manager` field points to an `struct Employee`.
- Only a self-referencing pointer is allowed, not an actual `struct Employee`.

Self-referencing example

```
struct Employee {
    char *firstName;
    char *lastName;
    struct Employee *manager;
};

void printEmployeeInfo( struct Employee *e );

int main()
{
    struct Employee boss = { "The", "Boss", NULL };
    struct Employee jane = { "Jane", "Doe", &boss };
    struct Employee john = { "John", "Doe", &boss };

    printEmployeeInfo( &boss );
    printEmployeeInfo( &jane );
    printEmployeeInfo( &john );
}
```

Self-referencing example

```
void printEmployeeInfo( const Employee *e )
{
    printf("First name: %s\n", e->firstName );
    printf("Last name: %s\n", e->lastName );
    printf("Manager: %s\n",
    (e->manager) ? e->manager->lastName
                 : "nobody" );
    puts("");
}
```

Self-referencing example

```
struct Word {
    char *word;
    char *definition;
    struct Word *synonyms[10];
};

void printWord( struct Word *w )
{
    printf("Word: %s\n",w->word);
    printf("Definition: %s\n",w->definition);
    printf("Synonyms:\n");
    for ( int i = 0; i < 10; i++ ) {
        if ( w->synonyms[i] == NULL ) break;
        printf("\t%s\n", w->synonyms[i]->word );
    }
}
```

Card shuffling example

```
typedef struct {
    const char *face;
    const char *suit;
} Card;

const char * kFaces[] = {
"Ace", "Deuce", "Three", "Four", "Five",
"Six", "Seven", "Eight", "Nine", "Ten",
"Jack", "Queen", "King" };

const char * kSuits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

void fillDeck( Card deck[] );
void shuffle( Card deck[] );
void deal( Card deck[] );
```

Card shuffling example

```
void fillDeck( Card deck[] )
{
    for ( int i = 0; i < 52; i++ )
    {
        deck[i].face = kFaces[i % 13];
        deck[i].suit = kSuits[i / 13];
    }
}
```

Card shuffling example

```
void shuffle( Card deck[] )
{
    for ( int i = 0; i < 52; i++ )
    {
        int j = rand() % 52;
        // swap cards
        Card temp = deck[i];
        deck[i] = deck[j];
        deck[j] = temp;
    }
}
```

Card shuffling example

```
void deal( Card deck[] )
{
    for ( int i = 0; i < 52; i++ )
    {
        printf("%s of %s\n", deck[i].face, deck[i].suit );
    }
}
```

Card shuffling example

```
int main()
{
    Card deck[52];

    srand(time(NULL));

    fillDeck( deck );
    shuffle( deck );
    deal( deck );
}
```

Unions

- A `union` is like a struct, except all member variables *share the same storage space!*

```
union Number {
    int integer;
    float real;
};
```

- Here, `integer` and `real` actually overlap in memory.
- `sizeof(union Number)` = 4 bytes.

Union example

```
union Number {
    int integer;
    float real;
};

int main() {
    union Number n;

    n.integer = 100;
    printf("as int: %d\n",n.integer);
    printf("as float: %f\n",n.real);
    puts("");

    n.real = 100.0;
    printf("as int: %d\n",n.integer);
    printf("as float: %f\n",n.real);
    puts("");
}
```

Unions

- A `union` might be useful to save space, if you need the flexibility to store one of many data types.
- Another use is to break an integer into bytes:

```
union UInt {
    unsigned int integer;
    unsigend char bytes[4];
};
```

- Here, `integer` and `bytes` overlap in memory.
- `sizeof(union Number)` = 4 bytes.

Union example

```c
#include <stdio.h>
union UInt {
    unsigned int integer;
    unsigned char bytes[4];
};

int main() {
    union UInt n;

    n.integer = 1;

    for ( int i = 0; i < 32; i++ )
    {
        printf("%16u %03d %03d %03d %03d\n", n.integer,
        n.bytes[0], n.bytes[1], n.bytes[2], n.bytes[3] );
        n.integer *= 2;
    }
}
```