Sample code to illustrate a simple data structure (struct contact)

```c
#include <stdio.h>
#include <string.h>

typedef struct 
{
    char lastname[25];
    char firstname[25];
    int areaCode;
    int phoneNumber;
} CONTACT;

int main(int argc, char *argv[]) 
{
    int i;
    CONTACT phonebook[100];
    strcpy(phonebook[1].lastname,"Papadopoulos");
    strcpy(phonebook[0].lastname,"Alvarado");
    phonebook[0].areaCode=phonebook[1].areaCode = 858;

    printf("sizeof(CONTACT): %d \n", sizeof(CONTACT));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    for (i = 0; i <= 1; i++)
        printf("phonebook[%d](lastname,areaCode) (%s,%d)\n",i,
                phonebook[i].lastname, phonebook[i].areaCode);
}
```
Three Syntax Variants

1. Anonymous structure typedef
2. Named structure typedef
3. Structure definition followed by typedef

1. `typedef struct {
   char lastname[25];
   char firstname[25];
   int areaCode;
   int phoneNumber;
} CONTACT;`

2. `typedef struct contact {
   char lastname[25];
   char firstname[25];
   int areaCode;
   int phoneNumber;
} CONTACT;`

3. `struct contact {
   char lastname[25];
   char firstname[25];
   int areaCode;
   int phoneNumber;
};

typedef struct contact CONTACT;`
Assignment with structures

• If a and b are valid data types, then b is copied into a
  • This is **different** than java
    • Java deals (almost exclusively) with Object references

• For example, in the previous code, If you added the statement
  • Then, the entire contents ( a CONTACT) in array location 1 would be copied to array location 2.

• C deals with references as a separate concept (pointers)
Function invocation/Function return

- Every subroutine is a function in C, and can return any valid datatype
  - this includes struct types

- One declares exactly what the method returns
  - Just like java

- See next slide for code the fuses several of these
Sample code that defines functions that return structs, copies structs

```c
#include <stdio.h>
#include <string.h>
#include "structure.h"

CONTACT formFill(char last[], int code)
{
    CONTACT rval;
    strcpy(rval.lastname, last);
    rval.areaCode = code;
    return rval;
}

void printbook(CONTACT book[], int nentries)
{
    int i;
    for (i = 0; i < nentries; i++)
        printf("book[%d](lastname,areaCode) (%s,%d)\n", i, book[i].lastname, book[i].areaCode);
}
... Continued next slide
```
Sample code that defines functions that return structs, copies structs

```c
int main(int argc, char *argv[])
{
    int i;
    CONTACT phonebook[100];
    phonebook[0] = formFill("Papadopoulos", 858);
    phonebook[1] = formFill("Alvarado", 858);
    phonebook[2] = phonebook[1]; /* copy entries */

    printf("sizeof(CONTACT): %d \n", sizeof(CONTACT));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    printbook(phonebook,3);

    /* Modify entry 2 last name */
    printf("Modify entry 2\n");
    phonebook[2] = formFill("Smith", 619);
    printbook(phonebook,3);
}
```

Call function to fill out a CONTACT, copy the returned data structure

Copy data structures, not references

Modify [2], then print again
Functions can return references (aka pointers)

• So far, we’ve explicitly shown returns of built-in types (e.g., void, int, ...) AND struct types (e.g. struct contact)

• But there are times, when one wants just a reference to a complex data structure, without a (deep) copy
    • C can do this

• Foreshadowing: if we wanted a function myFunc to return a reference to a CONTACT, one would declare
    • CONTACT * myFunc( ... )
    • The “*” means “pointer to”

• And then, only the pointer would be returned (Which is what Java does)
How are arguments passed to C?

• by VALUE!

• The compiler makes a COPY of what is passed in the argument list and places it on the stack.
  • This is what Java does with primitive types
  • This is NOT what Java does with Object types!
/* This does NOT modify the form of the caller */
void modifyForm(CONTACT form, char last[], int code)
{
    strcpy(form.lastname, last);
    form.areaCode = code;
}

int main(int argc, char *argv[])
{
    int i;
    CONTACT phonebook[100];
    phonebook[0] = formFill("Papadopoulos", 858);
    phonebook[1] = formFill("Alvarado", 858);
    phonebook[2] = phonebook[1]; /* copy entries */

    printf("sizeof(CONTACT): %d \n", sizeof(CONTACT));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    printbook(phonebook,3);

    /* Modify entry 2 last name */
    printf("(Improperly) Modify entry 2\n");
    modifyForm(phonebook[2], "Smith", 619);
    printbook(phonebook,3);
}
Pointers!

• What ARE pointers in C?
  • A variable that contains the memory address of another variable

• How do you get the address of an existing variable?
  • the ‘&’ operator
  • `int i; int * ipointer = &i;  /* ipointer has the memory address of i */`

• Can you get the memory address of a variable of primitive type?
  • YES!
  • You can even get the memory address of a variable that itself holds a memory address
    • So-called “a pointer to a pointer”
The return of memory models!!

• Consider the following lines of code:

```c
int i = 4;
int *p = &i;
```

What is the value of p?
A. 4  
B. The address of the variable i  
C. This code causes a compile error  
D. Other
The return of memory models!

• Consider the following lines of code:

```c
int i = 4;
int *p = &i;
```

What is the value of *p?

A. 4
B. The address of the variable i
C. This code causes a compile error
D. Other
Do not fear pointers! They play by rules... I promise

• Consider the following lines of code:

```c
int i = 4;
int *p = &i;
int **pp = &p;
```

What is the value of \&p?

A. 4  
B. The address of the variable i  
C. The address of the variable p  
D. Other
Do not fear pointers! They play by rules... I promise

- Consider the following lines of code:

```c
int i = 4;
int *p = &i;
int *p2 = &i;
int j = i;
p = 6;
```

What is the value of i at the end of this code?
- A. 4
- B. 6
- C. The address of p
- D. This code causes an error
Do not fear pointers! They play by rules... I promise

• Consider the following lines of code:

```c
int i = 4;
int *p = &i;
int *p2 = &i;
int j = i;
*p = 6;
```

What is the value of j at the end of this code?
A. 4  
B. 6  
C. The address of p  
D. This code causes an error
Do not fear pointers! They play by rules... I promise

• Consider the following lines of code:

```c
int i = 4;
int *p = &i;
int *p2 = &i;
int j = i;
p = 6;
```

What is the value of *p2 at the end of this code?
A. 4
B. 6
C. The address of p
D. This code causes an error
Pointers!

• What ARE pointers in C?
  • A variable that contains the memory address of another variable

• How do you get the address of an existing variable?
  • the ‘&’ operator
  • int i; int * ipointer = &i; /* ipointer has the memory address of i */

• Can you get the memory address of a variable of primitive type?
  • YES!
  • You can even get the memory address of a variable that itself holds a memory address
    • So-called “a pointer to a pointer”

• Let’s revisit our “failed” modifyForm() method of the previous
Sample code that will modify phonebook[2] (diff output)

```c
$ diff structure4.c structure5.c
18,19c18,19
< /* This does NOT modify the form of the caller */
< void modifyForm(CONTACT form, char last[], int code)
---
> /* This does modify the form of the caller */
> void modifyForm(CONTACT * form, char last[], int code)
21c21
<   strcpy(form.lastname, last);
<   form.areaCode = code;
---
>   strcpy(form->lastname, last);
>   form->areaCode = code;
37,38c37,38
<   printf("(Improperly) Modify entry 2\n");
<   modifyForm(phonebook[2], "Smith", 619);
---
>   printf("(properly) Modify entry 2\n");
>   modifyForm(________________, "Smith", 619);
```

Declare variable form to be a pointer to (memory address of) a CONTACT inside modifyForm. Copy what data into the fields using -> instead of . (to dereference pointer)

What code goes in the blank?
A. & (phonebook[2])
B. phonebook[2]
C. *(phonebook[2])
D. &phonebook
$ diff structure4.c structure5.c
18,19c18,19
< /* This does NOT modify the form of the caller */
< void modifyForm(CONTACT form, char last[], int code)
---
> /* This does modify the form of the caller */
> void modifyForm(CONTACT * form, char last[], int code)
21c21
<     strcpy(form.lastname, last);
<     form.areaCode = code;
---
>     strcpy(form->lastname, last);
>     form->areaCode = code;
37,38c37,38
<     printf("(Improperly) Modify entry 2\n");
<     modifyForm(phonebook[2], "Smith", 619);
---
>     printf("(properly) Modify entry 2\n");
>     modifyForm(&(phonebook[2]), "Smith", 619);

Sample code that will modify phonebook[2] (diff output)

Declare variable form to be a pointer to (memory address of) a CONTACT
inside modifyForm. Copy what data into the fields using -> instead of . (to dereference pointer)

 &: the address of phonebook[2]
Pointers and Arrays

• A pointer knows what kind of datatype it points to, e.g
  • int * intPointer;
  • double * floatPointer;
  • CONTACT * contactPointer;

• Arrays Know what kind of datatypes they hold
  • int intArray[30];
  • double doubleArray[80];
  • CONTACT phonebook[100];

• intArray (without brackets) is a pointer an int
  • Happens to be the first address of the indexed data structure (array of ints)
Pointers and Arrays Continued

• Equivalences (if intPointer = &(intArray[0]))
  • *intPointer == intArray[0]
  • *(intPointer +k) == intArray[k]
  • *(++intPointer) == intArray[1]

• Equivalences (if contactPointer = &(phonebook[0]))
  • *contactPointer == phonebook[0]
  • *(contactPointer +k) == phonebook[k]
  • *(++contactPointer) == phonebook[1]

• This is called POINTER ARITHMETIC.
  • ++, --, +=, -= all index the pointer to the proper memory location based upon the size of data structure being pointed to.
Since C passes arguments by value, how do you change a variable passed to the routine?

- Pass a POINTER to the function

```c
#include <stdio.h>
void square (int *iarg) {
    *iarg = (*iarg) * (*iarg);
}
main() {
    int i = 10;
    square(&i);
    printf(“%d\n”, i);
}
```

*`iarg`: dereference the pointer (get its actual contents)
*`iarg`: dereference the pointer (store in its memory)
`&i`: address of i (pointer to i)
printf(“100”)
prints “100”
decoding \texttt{int\ main(int\ argc,\ char\ \ast\ argv[])}

- \texttt{argc} – an integer describing the number of command line arguments passed to \texttt{main} by the command shell.
- \texttt{char \ast argv[]}
  - This takes some explaining
  - argv[] == char \ast argv \textarrow{\leftarrow} Pointer to characters (an array)
  - *argv[] == char \ast (*argv) = char **argv
    - pointer to an array of character arrays
    - Or in “Java-nese” an array of Strings.
- Commonly, you will see
  - \texttt{int\ main(int\ argc,\ char\ **argv)}
$ cat printArgs.c
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < argc; i++)
        printf("argv[%d]:'%s'\n", i, argv[i]);
}

$ cc -o printArgs printArgs.c
$ ./printArgs first "second" "third in quotes"
argv[0]:'./printArgs'
argv[1]:'first'
argv[2]:'second'
argv[3]:'third in quotes'

note that argv[0] is the name of the program itself (not the first command argument as in java)