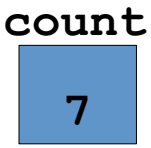
Lecture 09X: C Function Pointers

Concurrent and Multicore Programming

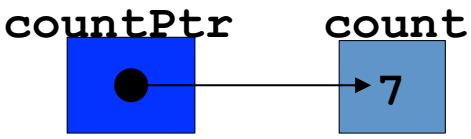
Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)
    - int count = 7;



  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value
    - int count = 7;
    - int * countPtr = &count;



Pointer Variable Declarations and Initialization

- Pointer declarations
  - **`*`** used with pointer variables

    ```
    int *myPtr;
    ```

  - Declares a pointer to an **`int`** (pointer of type **`int *`**)
  - Multiple pointers require using a **`*`** before each variable declaration

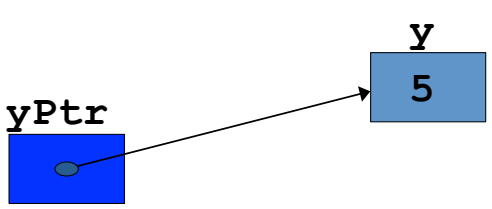
    ```
    int *myPtr1, *myPtr2;
    ```

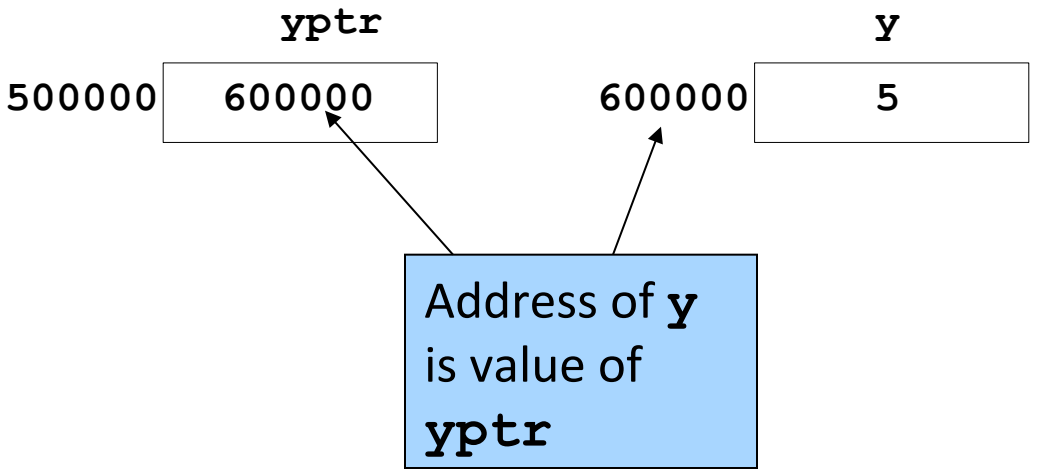
  - Can declare pointers to any data type
  - Initialize pointers to **`0`**, **`NULL`**, or an address
    - **`0`** or **`NULL`** – points to nothing (**`NULL`** preferred)

Pointer Operators

- **&** (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;      // yPtr gets address of y
yPtr "points to" y
```





Pointers and Arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Declare an array **`b[ 5 ]`** and a pointer **`bPtr`**
  - To set them equal to one another use:

    ```
    bPtr = b;
    ```

    - The array name (**`b`**) is actually the address of first element of the array **`b[ 5 ]`**

    ```
    bPtr = &b[ 0 ]
    ```

    - Explicitly assigns **`bPtr`** to address of first element of **`b`**

Pointers and Arrays

- Element `b[ 3 ]`
  - Can be accessed by `*( bPtr + 3 )`
    - Where `n` is the offset. Called pointer/offset notation
  - Can be accessed by `bptr[ 3 ]`
    - Called pointer/subscript notation
    - `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself

    `*( b + 3 )`

Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

Pointers to functions: Variable for functions

- Declaration:

```
returnType (*funVarName)(parameterTypes);
```

- Examples:

```
int (*f)(int, float);

int *(*g[])(int, float);

int *(*g[])(int, float);
```

pointer to a function that takes an integer argument and a float argument and returns an integer

pointer to a function that takes an integer argument and a float argument and returns a *pointer* to an integer

An *array* of pointers to functions – Each function takes an integer argument and a float argument and returns a pointer to an integer

Pointers to functions: WHY?

- They allow for a certain amount of **polymorphism**:
  - "poly" (many) + "morph" (shape)
  - A polymorphic language can handle a range of different data types ("shapes"?) with a single statement
- This is common in OO languages like C++, Java:

```
Animal myPet;
…
myPet.makeSound();
```

This method call will result in different sounds, depending on whether `myPet` holds a `Cow` object, an `Elephant` object, etc.

Example: searching a singly-linked list

```
typedef struct IntNode {
  int value;      struct IntNode *next;
} INTNODE;
```

OK, but it only works for nodes containing integer data. If you want a list of strings, you'll need to define a new type and new function.

```
INTNODE *search_list(INTNODE *node, int const key) {
  while (!node) {
   if (node->value == key) break;
   node = node->next;
  }
  return node;
}
```

A more abstract notion of "node"

```
typedef struct Node {
  void *value; struct Node *next;
} NODE;

void construct_node(NODE *node, void *value, NODE *next) {
  node->value = value;  node->next = next;
}

NODE *new_node(void *value, NODE *next) {
  NODE *node = (NODE *)malloc(sizeof(NODE));
  construct_node(node, value, next);
  return node;
}
```

`void*` is compatible with any pointer type.
So, this member can hold (a pointer to) any value!

A more abstract notion of "search list"

- What is it that makes the old `search_list` only work for integers?
  - The `key` parameter is of type `int`
  - The == operator is used to compare `int` values – but == will not work for many types (e.g. structs, strings)
- A solution: pass in an additional argument – a comparison function!
  - Programmer must supply a comparison function that's appropriate for the data type being stored in the nodes
  - This function argument is called a **callback function**:
    - Caller passes in a pointer to a function
    - Callee then "calls back" to the caller-supplied function

Abstract “search list” with callback function

```
NODE *search_list(NODE *node, void const *key,
    int (*compare)(void const *, void const *)) {

  while (node) {
   if (!compare(node->value, key)) break;
   node = node->next;
  }
  return node;

}
```

Assumption: `compare` returns zero if its parameter values are equal; nonzero otherwise

Using callback functions

- If our nodes hold strings, we have a compare function already defined: `strcmp` or `strncmpy`

```
#include <string.h>
…
match = search_list(root, "key", &strcmp);
```

& is optional here –
compiler will implicitly take the address

Note: you may get a warning, since `strcmp` is not strictly of the right type:
its parameters are of type `char *` rather than `void *`

Using callback functions

- If our nodes hold other kinds of data, we may need to “roll our own” compare function

```
int compare_ints(void const *a, void const *b) {
  const int ia = *(int *)a, ib = *(int *)b;
  return ia != ib;
}
…
match = search_list(root, key, &compare_ints);
```

Jump tables

- In some cases, a nice alternative to long, repetitive switch statements, like this:

```
double add(double, double);
double sub(double, double);
double mul(double, double);
double div(double, double);

switch(oper) {
case ADD:   result = add(op1, op2); break;
case SUB:   result = sub(op1, op2); break;
case MUL:   result = mul(op1, op2); break;
case DIV:   result = div(op1, op2); break;
}
```

Jump tables

- Jump table alternative:

```
double add(double, double);
double sub(double, double);
double mul(double, double);
double div(double, double);

double (*oper_func[])(double, double) = {
  add, sub, mul, div
};

result = oper_func[oper](op1, op2);
```

Array of pointers to functions.
Each function takes two `double`s
and returns a `double`

Pointers to functions: safety concerns

- What if uninitialized function pointer value is accessed?
  - Safest outcome: memory error, and program is terminated
  - But what if the "garbage" value is a valid address?
    - Worst case: address contains program instruction – execution continues, with random results
    - Hard to trace the cause of the erroneous behavior

References

- The Function Pointer Tutorials. http://www.newty.de/fpt/index.html