COMP26120: Pointers in C (2018/19)

Lucas Cordeiro
lucas.cordeiro@manchester.ac.uk
Organisation

• Lucas Cordeiro (Senior Lecturer, FM Group)
  – lucas.cordeiro@manchester.ac.uk
  – Office: 2.44
  – Office hours: 10-11 Tuesday, 2-3 Thursday

• Textbook:
  – C How to Program (chapter 7)

These slides are based on the lectures notes of “C How to Program”

©2016 Pearson Education, Inc., Hoboken, NJ. All rights reserved.
Intended learning outcomes

• To learn about **pointers** and **pointer operations**
• To use pointers to **pass arguments** to functions by **reference**
• To understand the close relationships among **pointers, arrays, and strings**
• To use **pointers to functions**
• To define and use **arrays of strings**
Introduction

- Pointers enable programs to
  - simulate *pass-by-reference*

```c
int x=2;
void square(int *nPtr){
    *nPtr *=*nPtr;
}
int main(void) {
    square (&x);
    return 0;
}
```

What are the **advantages** and **disadvantages** to pass arguments to functions by reference?
Introduction

• Pointers enable programs to
  – simulate pass-by-reference
  – use function pointers

```c
#include <stdio.h>
typedef void (*functiontype)(int x);
void dosomething(int x){
    printf("x: %d\n", x);
}
int main(void){
    functiontype func = &dosomething;
    func(2);
    return 0;
}
```

What is the common use of function pointers?
Introduction

• Pointers enable programs to
  – simulate pass-by-reference
  – use function pointers
  – create and manipulate **dynamic data structures**

```c
entry *allocate_entries(int num_entries){
    entry *entriesPtr = (entry*)malloc(sizeof(entry) * num_entries);
    for(int i = 0; i < num_entries; i++)
        init_entry(&entriesPtr[i]);
    return entriesPtr;
}
```

• Next class: dynamic memory management
Pointer Variable Definitions and Initialization

• **Pointers** are variables whose values are *memory addresses*
  – A variable directly contains a **specific value**
  – A pointer contains an **address of a variable** that contains a specific value
    • a variable name **directly** references a value, and a pointer **indirectly** references a value

• Referencing a value through a pointer is called **indirection**
Declaring Pointers

• Pointers, like all variables, must be defined before they can be used

• The definition
  
  • `int *countPtr, count;`
  • `void *speedPtr;`

  – How can we interpret these definitions?

• **Pointers** can be defined to **point to objects of any type**
Common Programming Error

• The asterisk (*) notation used to declare pointer variables does not distribute to all variable names in a declaration
  – Each pointer must be declared with the * prefixed to the name
    • xPtr and yPtr must be declared as *xPtr and *yPtr

• Include the letters ptr in pointer variable names
  – make it clear that these variables are pointers
  – need to be handled appropriately
Initialising and Assigning Values to Pointers

• Pointers should be initialised when they’re defined
  – A pointer may be initialised to NULL, 0 or an address
    • A pointer with the value NULL points to *nothing*
    • Initializing a pointer to 0 is equivalent to initialising a pointer to NULL, but NULL is preferred
  – NULL is a *symbolic constant* defined in the `<stddef.h>` header (and other headers, `<stdio.h>`)
Pointer Operators

• The &, or **address operator**, is a unary operator that returns the address of its operand

• For example, assuming the definitions
  
  ```c
  int y = 5;
  int *yPtr;
  ```

  the statement
  
  ```c
  yPtr = &y;
  ```

  assigns the **address** of the variable **y** to pointer variable **yPtr**

• Variable **yPtr** is then said to “point to” **y**
Pointer Representation in Memory

- Assume that integer variable \( y \) is stored at location 600000, and pointer variable \( yPtr \) is stored at location 500000

- The operand of the address operator must be a variable
  - the address operator cannot be applied to constants or expressions
The Indirection (*) Operator

• The unary * operator: indirection operator or dereferencing operator
  – returns the object value to which its operand points

• For example, the statement
  • printf("%d", *yPtr);
  prints the value of variable y, namely 5
  – Using * here is called dereferencing a pointer

Dereferencing a pointer that has not been initialised or that has not been assigned to point to a specific location in memory is an error
%p outputs the memory location as a hexadecimal integer on most platforms.
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0

Fig. 7.4  Using the & and * pointer operators. (Part 2 of 2.)

• The & and * operators are complements of one another—when they’re both applied consecutively to aPtr in either order, the same result is printed
<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>left to right</td>
<td>postfix</td>
</tr>
<tr>
<td>[]</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>++</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>--</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>-</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>*</td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td>/</td>
<td>left to right</td>
<td>logical OR</td>
</tr>
<tr>
<td>%</td>
<td>left to right</td>
<td>conditional</td>
</tr>
<tr>
<td>+</td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td>-</td>
<td>left to right</td>
<td>comma</td>
</tr>
<tr>
<td>&lt;</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>right to left</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 7.5**  | Precedence and associativity of the operators discussed so far.
Exercise

Answer each of the following:

a) A pointer variable contains as its value the ________ of another variable?

b) The three values that can be used to initialise a pointer are ______, ______ and ______.

c) The only integer that can be assigned to a pointer is ______.

d) The address operator can be applied only to ________.
Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function: **pass-by-value** and **pass-by-reference**
  - *All arguments in C are passed by value*

- Many functions require the capability to
  - modify variables in the **caller** or
  - pass a pointer to a large data object
    - To avoid the time and memory overheads

- In C, you use **pointers** and the **indirection** operator to **simulate** **pass-by-reference**
// Fig. 7.6: fig07_06.c
// Cube a variable using pass-by-value.
#include <stdio.h>

int cubeByValue(int n); // prototype

int main(void)
{
    int number = 5; // initialize number

    printf("The original value of number is %d", number);

    // pass number by value to cubeByValue
    number = cubeByValue(number);

    printf("\nThe new value of number is %d\n", number);
}

// calculate and return cube of integer argument
int cubeByValue(int n)
{
    return n * n * n; // cube local variable n and return result
}

Fig. 7.6  |  Cube a variable using pass-by-value. (Part 1 of 2.)
Use **call-by-value** to pass arguments to a function to prevent accidental modification of the caller’s argument.

The original value of number is 5
The new value of number is 125

**Fig. 7.6** Cube a variable using pass-by-value. (Part 2 of 2.)
// Fig. 7.7: fig07_07.c
// Cube a variable using pass-by-reference with a pointer argument.

#include <stdio.h>

void cubeByReference(int *nPtr); // function prototype

int main(void)
{
    int number = 5; // initialize number

    printf("The original value of number is %d", number);

    // pass address of number to cubeByReference
    cubeByReference(&number);

    printf("\nThe new value of number is %d\n", number);
}

// calculate cube of *nPtr; actually modifies number in main
void cubeByReference(int *nPtr)
{
    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}
The original value of number is 5
The new value of number is 125

**Fig. 7.7** Cube a variable using pass-by-reference with a pointer argument. (Part 2 of 2.)

**Use** pass-by-reference only if the caller explicitly requires the called function to modify the value of the argument variable in the caller’s environment.
Passing Arguments to Functions by Reference (Cont.)

- For a function that expects one-dimensional array, its prototype and header can use pointer notation
  - No difference between a function that receives a pointer and one that receives a one-dimensional array
  - the function must “know” when it’s receiving an array or a single variable to perform pass-by-reference

- The compiler converts int b[] to the pointer notation int *b
Fig. 7.8 | Analysis of a typical pass-by-value. (Part 1 of 3.)
Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:

```c
int main(void)
{
    int number = 5;
    number = cubeByValue(number);
}
```

```c
int cubeByValue(int n)
{
    return n * n * n;
}
```

Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

```c
int main(void)
{
    int number = 5;
    number = cubeByValue(number);
}
```

```c
int cubeByValue(int n)
{
    return n * n * n;
}
```

Fig. 7.8 | Analysis of a typical pass-by-value. (Part 2 of 3.)
Step 5: After `main` completes the assignment to `number`:

```c
int main(void)
{
    int number = 5;
    number = cubeByValue(number);
}
```

```c
int cubeByValue(int n)
{
    return n * n * n;
}
```

**Fig. 7.8** Analysis of a typical pass-by-value. (Part 3 of 3.)
Fig. 7.9  |  Analysis of a typical pass-by-reference with a pointer argument. (Part 1 of 2.)
Step 3: After `*nPtr` is cubed and before program control returns to `main`:

```c
int main(void)
{
    int number = 5;
    cubeByReference(&number);
}

void cubeByReference(int *nPtr)
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

*Fig. 7.9* | Analysis of a typical pass-by-reference with a pointer argument. (Part 2 of 2.)
sizeof Operator

• C provides the special unary operator `sizeof` to determine the size in bytes of an array (or any other data type)
• `sizeof` is a compile-time operator, so it does not incur any execution-time overhead
// Fig. 7.16: fig07_16.c
// Applying sizeof to an array name returns
// the number of bytes in the array.
#include <stdio.h>
define SIZE 20

size_t getSize(float *ptr); // prototype

int main(void)
{
    float array[SIZE]; // create array

    printf("The number of bytes in the array is %u\n"
           "The number of bytes returned by getSize is %u\n",
           sizeof(array), getSize(array));
}

// return size of ptr
size_t getSize(float *ptr)
{
    return sizeof(ptr);
}
When applied to the name of an array, the `sizeof` operator returns the total number of bytes in the array as type `size_t`.

Variables of type `float` on this computer are stored in 4 bytes of memory, and array is defined to have 20 elements.

Therefore, there are a total of 80 bytes in array.
// Fig. 7.17: fig07_17.c
// Using operator sizeof to determine standard data type sizes.
#include <stdio.h>

int main(void)
{
    char c;
    short s;
    int i;
    long l;
    long long ll;
    float f;
    double d;
    long double ld;
    int array[20]; // create array of 20 int elements
    int *ptr = array; // create pointer to array

    printf("    sizeof c = %u\tsizeof(char) = %u"
        "\n    sizeof s = %u\tsizeof(short) = %u"
        "\n    sizeof i = %u\tsizeof(int) = %u"
        "\n    sizeof l = %u\tsizeof(long) = %u"
        "\n    sizeof ll = %u\tsizeof(long long) = %u"
        "\n    sizeof f = %u\tsizeof(float) = %u"");
```c
    
    sizeof d = %u \tsizeof(double) = %u
    
    sizeof ld = %u \tsizeof(long double) = %u
    
    sizeof array = %u
    
    sizeof ptr = %u

    sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i,
    sizeof(int), sizeof l, sizeof(long), sizeof ll,
    sizeof(long long), sizeof f, sizeof(float), sizeof d,
    sizeof(double), sizeof ld, sizeof(long double),
    sizeof array, sizeof ptr);
```
Exercise

• Consider the following array definition:
  • `double real[22];`
• What is the total number of bytes?

• How can you determine the total number of elements?
Pointer Expressions and Pointer Arithmetic

• **Pointers** are valid operands in arithmetic expressions, assignment expressions and comparison expressions

• A pointer may be
  
  – *incremented* (++) or *decremented* (--)
  
  – an integer may be *added* to a pointer (+ or +)=
  
  – an integer may be *subtracted* from a pointer (- or -=)
  
  – one pointer may be subtracted from another if *both* pointers point to elements of the *same* array
Since the results of pointer arithmetic depend on the size of the objects a pointer points to, **pointer arithmetic is machine dependent**.
Pointer Arithmetic Example

• When an integer is added to or subtracted from a pointer, the pointer is *not* incremented or decremented simply by that integer,
  – By that integer times the size of the object to which the pointer refers
    • The number of bytes depends on the object’s data type

• What this statement would produce?
  • vPtr += 2;
    – it would produce 3008 (3000 + 2 * 4), assuming an integer is stored in 4 bytes of memory
    – In the array v, vPtr would now point to v[2]
Using pointer arithmetic on a pointer that does not refer to an element in an array is a common programming error.
A pointer can be assigned to another pointer if both have the same type
- The exception to this rule is the pointer to void (i.e., `void *`), which is a generic pointer that can represent any pointer type

All pointer types can be assigned a pointer to `void`, and a pointer to `void` can be assigned a pointer of any type
- In both cases, a cast operation is not required

A pointer to void cannot be dereferenced
Pointer Expressions and Pointer Arithmetic (Cont.)

• The compiler knows that a pointer to `int` refers to 4 bytes of memory on a machine with 4-byte integers
  – however, a pointer to `void` simply contains a memory location for an *unknown* data type
  – the precise number of bytes to which the pointer refers is not known by the compiler

The compiler *must* know the data type to determine the number of bytes to be dereferenced for a particular pointer
Pointer Expressions and Pointer Arithmetic (Cont.)

• Pointers can be compared using equality and relational operators, but such comparisons need to point to elements of the same array
  – Pointer comparisons compare the addresses stored in the pointers

• A comparison of two pointers pointing to elements in the same array could show that one pointer points to a higher-numbered element of the array than the other pointer does

• A common use of pointer comparison is determining whether a pointer is NULL
Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably
  - An *array name* can be thought of as a **constant pointer**
  - **Pointers** can be used to do any operation involving array indexing
- Assume that integer array `b[5]` and integer pointer variable have been defined
  - Attempting to modify an array name with pointer arithmetic is a compilation error (e.g., `b+=3`)
// Fig. 7.20: fig07_20.cpp
// Using indexing and pointer notations with arrays.
#include <stdio.h>
#define ARRAY_SIZE 4

int main(void)
{
    int b[] = {10, 20, 30, 40}; // create and initialize array b
    int *bPtr = b; // create bPtr and point it to array b

    // output array b using array index notation
    puts("Array b printed with:\nArray index notation");

    // loop through array b
    for (size_t i = 0; i < ARRAY_SIZE; ++i) {
        printf("b[%u] = %d\n", i, b[i]);
    }

    // output array b using array name and pointer/offset notation
    puts("\nPointer/offset notation where
    "the pointer is the array name");
}
// loop through array b
for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
    printf("*(b + %u) = %d\n", offset, *(b + offset));
}

// output array b using bPtr and array index notation
put("\nPointer index notation");

// loop through array b
for (size_t i = 0; i < ARRAY_SIZE; ++i) {
    printf("bPtr[%u] = %d\n", i, bPtr[i]);
}

// output array b using bPtr and pointer/offset notation
put("\nPointer/offset notation");

// loop through array b
for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
    printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
}
Array b printed with:
Array index notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer index notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

Fig. 7.20  |  Using indexing and pointer notations with arrays. (Part 3 of 3.)
Exercise

• Find the error in each of the following program segments:

```c
int *zPtr;
int *aPtr = NULL;
void *sPtr = NULL;
int number, i;
int z[5] = {1, 2, 3, 4, 5};
sPtr = z;
```

a) `++zPtr;`
b) `number = zPtr;`
c) `number = *zPtr[2];`
d) `number = *sPtr;`
e) `++z;`
Using the `const` Qualifier with Pointers

- The `const` qualifier indicates that the value of a particular variable should not be modified
  - Reduce debugging time and improper side effects
    - It also makes a program easier to modify and maintain
  - Ensure that data is **not accidentally modified**

If an attempt is made to modify a value that’s declared `const`, the compiler catches it and issues either a **warning** or an **error**
Using the `const` Qualifier with Pointers (Cont.)

Access privileges

- Non-Constant Pointer
- Constant Pointer
- Non-Constant Data
- Constant Data

Which combination provides the **highest level of data access**?
// Fig. 7.10: fig07_10.c
// Converting a string to uppercase using a
// non-constant pointer to non-constant data.
#include <stdio.h>
#include <ctype.h>

void convertToUppercase(char *sPtr); // prototype

int main(void)
{
    char string[] = "ChArAc ters and $32.98"; // initialize char array

    printf("The string before conversion is: %s", string);
    convertToUppercase(string);
    printf("\nThe string after conversion is: %s\n", string);
}

Fig. 7.10  |  Converting a string to uppercase using a non-constant pointer to non-constant
data. (Part 1 of 2.)
// convert string to uppercase letters
void convertToUpper(char *sPtr)
{
    while (*sPtr != '\0') { // current character is not '\0'
        *sPtr = toupper(*sPtr); // convert to uppercase
        ++sPtr; // make sPtr point to the next character
    }
}

The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98

Fig. 7.10  |  Converting a string to uppercase using a non-constant pointer to non-constant data. (Part 2 of 2.)
Example of Non-Constant Pointer to Constant Data

- A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified
Fig. 7.11: Printing a string one character at a time using a non-constant pointer to constant data. (Part 1 of 2.)
19    // sPtr cannot be used to modify the character to which it points,
20    // i.e., sPtr is a "read-only" pointer
21    void printCharacters(const char *sPtr)
22    {
23        // loop through entire string
24        for (; *sPtr != '\0'; ++sPtr) { // no initialization
25            printf("%c", *sPtr);
26        }
27    }

The string is:
print characters of a string

**Fig. 7.11**  | Printing a string one character at a time using a non-constant pointer to constant data. (Part 2 of 2.)
Attempt to Modify Data via a Non-Constant Pointer to Constant Data

• The next example illustrates the attempt to compile a function that receives a non-constant pointer (xPtr) to constant data
• This function attempts to modify the data pointed to by xPtr—which results in a compilation error
// Fig. 7.12: fig07_12.c
// Attempting to modify data through a
// non-constant pointer to constant data.
#include <stdio.h>
void f(const int *xPtr); // prototype

int main(void)
{
    int y; // define y

    f(&y); // f attempts illegal modification
}

// xPtr cannot be used to modify the
// value of the variable to which it points
void f(const int *xPtr)
{
    *xPtr = 100; // error: cannot modify a const object
}

error C2166: l-value specifies const object

Fig. 7.12  |  Attempting to modify data through a non-constant pointer to constant data.
Trade-off Memory and Execution Efficiency

- If memory is low and execution efficiency is a concern, use pointers.
- If memory is in abundance and efficiency is not a major concern, pass data by value to enforce the principle of least privilege.
- Remember that some systems do not enforce const well, so pass-by-value is still the best way to prevent data from being modified.
Attempts to Modify a Constant Pointer to Non-Constant Data

- A constant pointer to non-constant data always points to the same memory location
  - the data at that location can be modified through the pointer
  - This is the default for an array name (constant pointer)
- All data in the array can be accessed and changed by using the array name and array indexing

```c
int a[5];
for(int i=0; i<5; i++)
a[i] = 2*i;
```

Invalid operands to binary expression ('const int [5]' and 'int')
// Fig. 7.13: fig07_13.c
// Attempting to modify a constant pointer to non-constant data.
#include <stdio.h>

int main(void)
{
    int x; // define x
    int y; // define y

    // ptr is a constant pointer to an integer that can be modified
    // through ptr, but ptr always points to the same memory location
    int * const ptr = &x;

    *ptr = 7; // allowed: *ptr is not const
    ptr = &y; // error: ptr is const; cannot assign new address
}
Attempting to Modify a Constant Pointer to Constant Data

• The least access privilege is granted by a constant pointer to constant data

• Such a pointer always points to the same memory location, and the data at that memory location cannot be modified

```c
const int a[5];

for(int i=0; i<5; i++)
    printf("a[%i]: %i\n", i, a[i]);
```

```c
a[1]=2
```

read-only variable is not assignable
// Fig. 7.14: fig07_14.c
// Attempting to modify a constant pointer to constant data.
#include <stdio.h>

int main(void)
{
    int x = 5; // initialize x
    int y; // define y

    // ptr is a constant pointer to a constant integer. ptr always
    // points to the same location; the integer at that location
    // cannot be modified
    const int *const ptr = &x; // initialization is OK

    printf("%d\n", *ptr);
    *ptr = 7; // error: *ptr is const; cannot assign new value
    ptr = &y; // error: ptr is const; cannot assign new address
}
Summary

• Pointer variable definition and initialization
• Pointer operators
• Passing arguments to functions by reference
• Using the const qualifier with pointers
• sizeof operator
• Pointer expression and pointer arithmetic
• Relationship between pointers and arrays