Computer Programming 143 – Lecture 20
Pointers III

Electrical and Electronic Engineering Department
University of Stellenbosch

Prof Johan du Preez
Mr Callen Fisher
Dr Willem Jordaan
Dr Hannes Pretorius
Mr Willem Smit

Copyright & Disclaimer

Copyright

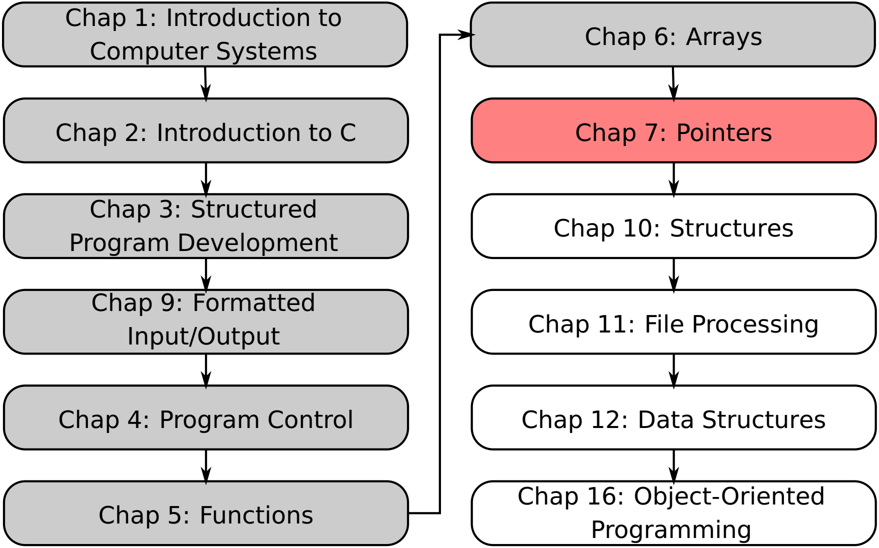
Disclaimer

This content is provided without warranty or representation of any kind. The use of the content is entirely at your own risk and Stellenbosch University (SU) will have no liability directly or indirectly as a result of this content.

The content must not be assumed to provide complete coverage of the particular study material. Content may be removed or changed without notice.

The video is of a recording with very limited post-recording editing. The video is intended for use only by SU students enrolled in the particular module.

Module Overview



Chap 1: Introduction to Computer Systems

Chap 2: Introduction to C

Chap 3: Structured Program Development

Chap 9: Formatted Input/Output

Chap 4: Program Control

Chap 5: Functions

Chap 6: Arrays

Chap 7: Pointers

Chap 10: Structures

Chap 11: File Processing

Chap 12: Data Structures

Chap 16: Object-Oriented Programming

Lecture Overview

1. Using the `const` Qualifier with Pointers (7.5)

2. Bubble Sort Using Call-by-Reference (7.6)

3. `sizeof` Operator (7.7)

7.5 Using the const Qualifier with Pointers I

const qualifier

- Variable cannot be changed

```
const double PI = 3.141592653589793238;
```

- Use const if function does not need to change a variable
- Attempting to change a const variable produces a compiler error

7.5 Using the const Qualifier with Pointers II

const qualifier with pointers

There are 4 possible combinations:

1. Non-constant pointer to non-constant data

   ```
   int *myPtr;
   ```

   - Both the value (address) of myPtr and the integer it points to (*myPtr) may be changed

2. Non-constant pointer to constant data

   ```
   const int *myPtr;
   ```

   - The value (address) of myPtr may be changed, but the integer it points to (*myPtr) may not

7.5 Using the const Qualifier with Pointers III

const qualifier with pointers

3. Constant pointer to non-constant data

```
int * const myPtr = &x;
```

   - The value (address) of myPtr may not be changed, but the integer it points to (*myPtr) may – myPtr points to an unchanging memory position
   - The value (address) of myPtr must be initialised at declaration

4. Constant pointer to constant data

```
const int * const myPtr = &x;
```

   - Not the value (address) of myPtr nor the integer it points to (*myPtr) may be changed
   - The value (address) of myPtr must be initialised at declaration

7.5 Using the const Qualifier with Pointers IV

When to use which combination?

- Applicable to argument declaration when writing function definitions
- Use the *principle of least privilege*
- Prevents errors – ensures that your function does not accidentally alter data

7.5 Using the const Qualifier with Pointers V

Example: non-constant pointer to non-constant data (Fig.7.10)

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

void convertToUppercase( char *sPtr ); // prototype

int main( void )
{
   char string[] = "characters and $32.98"; // initialise char array

   printf( "The string before conversion is: %s", string );
   convertToUppercase( string );
   printf( "\nThe string after conversion is: %s\n", string );
   return 0; // indicates successful termination
} // end main
```

7.5 Using the const Qualifier with Pointers VI

...Example: non-constant pointer to non-constant data

```
// convert string to uppercase letters
void convertToUppercase( char *sPtr )
{
   while ( *sPtr != ’\0’ ) { // current character is not ’\0’

      if ( islower( *sPtr ) ) { // if character is lowercase
         *sPtr = toupper( *sPtr ); // convert to uppercase
      } // end if

      ++sPtr; // move sPtr to the next character
   } // end while
} // end function convertToUppercase
```

7.5 Using the const Qualifier with Pointers VII

Output

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

Discussion of example

- `void convertToUppercase( char *sPtr )`
  - Passes a non-constant pointer to a non-constant character array as argument
- Library `ctype.h` contains character classification and manipulation functions
  - Function `islower()` tests if its argument is a lowercase character
  - Function `toupper()` returns the uppercase character of its argument
- `++sPtr` moves to the pointer to the next character (next lecture)

7.5 Using the const Qualifier with Pointers VIII

Example: non-constant pointer to constant data (Fig.7.11)

```c
#include <stdio.h>

void printCharacters( const char *sPtr );

int main( void )
{
   // initialise char array
   char string[] = "print characters of a string";

   printf( "The string is:\n" );
   printCharacters( string );
   printf( "\n" );
   return 0; // indicates successful termination
} // end main
```

7.5 Using the const Qualifier with Pointers IX

...Example: non-constant pointer to constant data

```
/* sPtr cannot modify the characters to which it points,
 * i.e., sPtr is a "read-only" pointer */
void printCharacters( const char *sPtr )
{
   // loop through entire string
   while (*sPtr != ’\0’) { // no initialisation
        printf( "%c", *sPtr );
        sPtr++;
   } // end for
} // end function printCharacters
```

Output

```
The string is:
print characters of a string
```

7.5 Using the const Qualifier with Pointers X

Refer to Fig. 7.12-7.14 in Deitel & Deitel for more examples of `const` and pointers

7.6 Bubble Sort Using Call-by-Reference I

```
void swap( int *element1Ptr, int *element2Ptr ); // prototype
// sort an array of integers using bubble sort algorithm
// could also use ‘int array[]‘ instead of ‘int * const array‘ below
void bubbleSort( int * const array, const int size ) {
   int pass; // pass counter
   int j; // comparison counter

   // loop to control passes
   for ( pass = 0; pass < size - 1; pass++ ) {
      // loop to control comparisons during each pass
      for ( j = 0; j < size - 1; j++ ) {
         // swap adjacent elements if they are out of order
         if ( array[ j ] > array[ j + 1 ] ) {
            swap( &array[ j ], &array[ j + 1 ] );
         } // end if
      } // end inner for
   } // end outer for
} // end function bubbleSort
```

7.6 Bubble Sort Using Call-by-Reference II

```c
/* swap values at memory locations to which element1Ptr and
 * element2Ptr point  */
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} // end function swap
```

Refer to Fig. 7.15 in Deitel & Deitel for the full program listing

7.7 sizeof Operator

sizeof

- Returns size of operand in bytes
- For arrays: size of 1 element × number of elements
- if sizeof(int) equals 4 bytes, then

```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

will print 40

sizeof can be used with

- Variable names
- Type name
- Constant values

```c
/*   Demonstrating the sizeof operator */
#include <stdio.h>

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

   printf( "\n sizeof c            = %d", sizeof( c ));
   printf( "\t sizeof(char)        = %d", sizeof( char ));
   printf( "\n sizeof s            = %d", sizeof( s ));
   printf( "\t sizeof(short)       = %d", sizeof( short ));
   printf( "\n sizeof i            = %d", sizeof( i ));
   printf( "\t sizeof(int)         = %d", sizeof( int ));
   printf( "\n sizeof l             = %d",  sizeof( l ));
   printf( "\t sizeof(long)         = %d", sizeof( long ));
   printf( "\n sizeof f             = %d",  sizeof( f ));
   printf( "\t sizeof(float)        = %d", sizeof( float ));

   printf( "\n sizeof d             = %d", sizeof( d ));
   printf( "\t sizeof(double)       = %d", sizeof( double ));
   printf( "\n sizeof ld            = %d", sizeof( ld ));
   printf( "\t sizeof(long double) = %d", sizeof( long double ));

   printf( "\n sizeof array         = %d", sizeof( array ));
   printf( "\t   sizeof ptr         = %d", sizeof( ptr ));
   printf("\n");

   return 0;
}
```

```
sizeof c      = 1     sizeof(char)        = 1
sizeof s      = 2     sizeof(short)       = 2
sizeof i      = 4     sizeof(int)         = 4
sizeof l      = 4     sizeof(long)        = 4
sizeof f      = 4     sizeof(float)       = 4
sizeof d      = 8     sizeof(double)      = 8
sizeof ld     = 12    sizeof(long double) = 12
sizeof array  = 80    sizeof ptr          = 4
```

sizeof and arrays

- sizeof only works on arrays inside the scope where the array is defined.
- If an array is passed to a function, sizeof cannot determine the memory size used by the array.
- sizeof(array) returns amount of memory consumed by all array elements.

Perspective

Today

Pointers III

- Using the `const` qualifier with pointers
- Bubble sort using call-by-reference
- `sizeof` operator

Next lecture

Pointers IV

- Pointer arithmetic
- Pointers and arrays

Homework

1. Study Sections 7.5-7.7 in Deitel & Deitel
2. Do Self Review Exercise 7.6 in Deitel & Deitel
3. Do Exercises 7.11, 7.19 in Deitel & Deitel