Lecture Notes for Lecture 8 of CS 5001 (Foundations of CS) for the Fall, 2018 session at the Northeastern University Silicon Valley Campus.

Introduction to Types and Using Typedefs to Define New Types

Philip Gust,
Clinical Instructor
Department of Computer Science

Lecture 7 Review

- An array element can be addressed using either an array index or a pointer to the array element.
- Array variables point to their first element; other elements can be accessed using address arithmetic and value-of ('*') operator.
- Incrementing array pointer takes size of element type into account (e.g. adds 4 bytes for an integer array).
- 2D arrays require expensive calculation to skip over previous rows, and then skip to column (array + row * numCols + col)
- Pointers are often more efficient to access arrays by avoiding repeated address calculation if access pattern is linear.
- Arrays are passed to functions using their base pointer; compiler still allows access using array indexes in certain cases
- Most C array and string library functions have pointer parameters.

Type Definitions

What is a Data Type?

- A data type is how a programming language classifies a set of values into a common category.
- A data type also defines how values can be manipulated, and how they interact with other data types.
- The C programming language defines a set of basic data types that include **bool**, **char**, **short**, **int**, **long**, **float**, and **double**.
- Operators and functions in C provide ways to operate on its basic data types, and cast (convert) between data types.

Type Definitions

What is a Data Type?

- As we begin to build programs to solve real-world problems, we need to deal with more complex kinds of information.
- For example, a graphics program provides geometrical shapes and ways to manipulate them and to give colors to them.
- It would be useful if we could create custom data types that represent shapes and colors, and provide custom functions that operate on them.

Type Definitions

Defining Custom Data Types

- The C programming language provides a set of tools for defining custom data types and creating functions that operate on them.
- These tools fall into five categories:
  - **References** provide indirect access to values of another data type.
  - **Sequences** are indexable homogeneous collections of another data type.
  - **Aliases** provide an alternate name for another data type
  - **Enumerations** explicitly identify collections of discrete values that comprise data types.
  - **Aggregations** are heterogeneous combinations of other data types.

Type Definitions

Defining Custom Data Types

- We have already seen two of the tools that C provides for creating custom data types.
- A pointer creates a custom data type that indirectly references another data type. A pointer to integer (int *) is a data type. Its operators are address-of (&) and value-of (*).
- An array creates a custom data type that is an indexable sequences of another data type. An array of int (int[]) is a data type. Its operator ([]) accesses a value in the sequence.

Type Definitions

Aliasing a Type

- This lecture presents a tool in the C language for defining a data type as an alias for an exiting data type: *typedef*.
- Typedefs are used to simplify references to more complex data types by providing descriptive names for those types.
- Typedefs can also be used to define size-specific data types that are portable across computer platforms
- Syntax:

  typedef *existing-type new-type*

Type Definitions

Portability Typedefs

- The C language provides typedef aliases for integer storage classes in terms of basic data types.
- These typedefs provide definitions for integer types with exactly 8, 16, 32, or 64 bits, either signed or unsigned.
- Compilers define them appropriately for the operating systems and the hardware. The definitions are in <stdlib.h>
- Why use them instead of the standard types?
    - Performing calculations that require a specific integer range
    - Writing data files where sizes are required by a standard file format
    - Communicating with other systems that require specific sizes

Type Definitions

Portability Typedefs

- Here are the ones for a typical 64-bit system:

```
typedef signed char int8_t;             // 8-bit signed  (INT8_MIN ... INT8_MAX)
typedef unsigned char uint8_t;          // 8-bit unsigned  (0 ... UINT8_MAX)

typedef short int16_t;                  // 16-bit signed  (INT16_MIN ... INT16_MAX)
typedef unsigned short uint16_t;        // 16-bit unsigned  (0 … UINT16_MAX)

typedef int int32_t;                    // 32-bit signed  (INT32_MIN … INT32_MAX)
typedef unsigned int uint32_t;          // 32-bit unsigned  (0 … UINT32_MAX)

typedef long int64_t;                   // 64-bit signed  (INT64_MIN … INT64_MAX)
typedef unsigned long uint64_t;         // 64-bit unsigned  (0L … UINT64_MAX)

typedef unsigned long size_t;           //  type of array size/index
```

- Examples using size-specific typedefs

```
size_t  slen = strlen(”hello");         // length of string is of type size_t
uint8_t  pixelVal = 0x6c;               // pixel defined as a 8-bit unsigned value
uint32_t  colorVal = 0xAB0D9F27;        // RGBA color defined as 32-bit unsigned value
```

Type Definitions

Defining a Boolean Type

- Before type *bool* was introduced in C99, developers sometimes used typedef to define their own bool type:

```
typedef uint8_t bool;            // represent bool using an 8-bit unsigned int
const uint8_t true= 1;           // false defined as the inequality value (0)
const uint8_t false = 0;         // true defined as the equality value (1)

/**
  * Return bool value true if input value is even, true otherwise
  * @param val the input value
  * @return true if even, false otherwise
  */
bool isEven(int val) {
   return (val % 2 ==0) ? true: false;
}
```

Type Definitions

Defining a Tic-Tac-Toe Board Type

- Typedef can be used to define a type for a Tic-Tac-Toe board:

```
/** TTTBoard is typedef for a 3x3 array of char */
typedef char TTTBoard[3][3];

/**
 * Initialize the tic-tac-toe board.
 * @param board the tic-tac-toe board
 */
void initBoard(TTTBoard board) {  …  board[row][col] = ‘ ‘;  …  }

/** Play tic-tac-toe */
int main(void) {
    TTTBoard board;         // declare tic-tac-toe board
    initBoard(board);       // function initializes board
    …
}
```

Type Definitions

Representing Colors

- Typedefs can be used to define types that encapsulate the underlying representation.
- For example, a color can be specified using four bytes for red (R), green (G), blue (B) components, plus a byte for alpha (A) transparency.

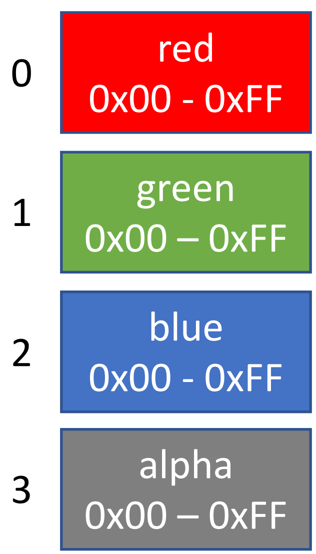
red	green	blue	alpha
0x00 - 0xFF	0x00 – 0xFF	0x00 - 0xFF	0x00 – 0xFF

RGBA image with transparent portions, on checkerboard background

Type Definitions

Representing Colors

- One way to represent a color is as an array of four uint8_t color values, with red as the lowest element 0, green as element 1, blue as element 2, and alpha as element 3.
- This order follows the OpenGL computer graphics standard.



Type Definitions

Representing Colors

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>

/** Color representation for components r, g, b, a */
typedef uint8_t Color[4];

/** Create a color from r, g, b, and a components.
 * @param c the color to set
 * @param r red color component
 * @param g green color component
 * @param b blue color component
 * @param a alpha color component
 */
void makeColorFromRGBA(Color c, uint8_t r, uint8_t g, uint8_t b, uint8_t a) {
    c[0] = r;
    c[1] = g;
    c[2] = b;
    c[3] = a;
}
```

Type Definitions

Representing Colors

```
/**
 * Get red color component.
 * @param color the color
 * @return red color component
 */
uint8_t getRedOfColor(const Color c) {
	return c[0];
}

/**
 * Get green color component.
 * @param color the color
 * @return green color component
 */
uint8_t getGreenOfColor(const Color c) {
	return c[1];
}
```

Type Definitions

Representing Colors

```
/**
 * Get blue color component.
 * @param color the color
 * @return blue color component
 */
uint8_t getBlueOfColor(const Color c) {
	return c[2];
}

/**
 * Get alpha color component.
 * @param color the color
 * @return alpha color component
 */
uint8_t getAlphaOfColor(const Color c) {
	return c[3];
}
```

Type Definitions

Representing Colors

- This representation has both advantages and disadvantages.
- Advantages:
  - The representation is straight-forward and easy to work with.
- Disadvantages:
  - A Color cannot be created and returned within the makeColor() function. Instead it must be created and passed in by the caller.
  - A Color cannot easily be copied, assigned, or compared to another color using standard assignment and relational operators in the same way as a basic C type.

Type Definitions

Representing Colors

- The first disadvantage is because an array declared within a function is local to the function and no longer exists after returning from the function. It must be passed in instead.
- C does not support an array return type. Return type would have to be declared as uint8_t *.
- Consequently, the following code would not compile or run.

```
Color makeColorFromRGBA(uint8_t r, uint8_t g, uint8_t b, uint8_t a) {
    Color c;    // local uint8_t[4] array no longer exists after return
    c[0] = r;
    c[1] = g;
    c[2] = b;
    c[3] = a;
    return c;   // cannot return array from function; c is a uint8_t*
}
```

Type Definitions

Representing Colors

- The second disadvantage can be addressed by providing a makeColorFromColor() function that initializes a Color from another Color.

```
/**
 * Create a color from the components of another color
 * @param c the color to set
 * @param c2 another color
*/
void makeColorFromColor(Color c, const Color c2) {
    c[0] = c2[0];
    c[1] = c2[1];
    c[2] = c2[2];
    c[3] = c2[3];
}
```

Type Definitions

Representing Colors

- An equalsColor() function would allow comparing two colors.

```
/**
 * Determine whether two colors are equal.
 * @param color1 the first color
 * @param color2 the second color
 * @return true if the two colors are equal
 */
bool equalsColor(const Color color1, const Color color2) {
    return          color1[0] == color2[0]
                 && color1[1] == color2[1]
                 && color1[2] == color2[2]
                 && color1[3] == color2[3];
}
```

Type Definitions

Representing Colors

- Here is a main function to test our new Color data type.

```
/** Test Color data type and functions */
int main(void) {
    Color c1;
    makeColorFromRGBA(c1, 0xFF, 0xF0, 0x0F, 0x01);
    uint8_t r1 = getRedOfColor(c1);
    uint8_t g1 = getGreenOfColor(c1);
    uint8_t b1 = getBlueOfColor(c1);
    uint8_t a1 = getAlphaOfColor(c1);
    printf( “Color c1:”
            ” r: 0x%02x”        // %02x specifies 2-digit 0-padded hex field
            " g: 0x%02x"
            " b: 0x%02x"
            " a: 0x%02x\n", r1, g1, b1, a1);
```

Type Definitions

Representing Colors

- Here is a main function to test our new Color data type.

```
    Color c2;
    makeColorFromColor(c2, c1);
    uint8_t r12= getRedOfColor(c2);
    uint8_t g2 = getGreenOfColor(c2);
    uint8_t b2 = getBlueOfColor(c2);
    uint8_t a2 = getAlphaOfColor(c2);
    printf( “Color c2:”
            ” r: 0x%02x"
            " g: 0x%02x"
            " b: 0x%02x"
            " a: 0x%02x\n", r2, g2, b2, a2);

    // determine whether c1 and c2 are equal
    bool eq = equalsColor(c1,c2);
    printf("Color c1 == c2? %s", eq ? "true" : "false");
}
```

Type Definitions

Representing Colors

- Another way to represent a color is to pack the four color bytes into a uint32_t
- On a little-endian computer, the bytes are arranged like this to conform to the OpenGL graphics standard order:

31 24	23 16	15 8	7 0
alpha 0x00 – 0xFF	blue 0x00 - 0xFF	green 0x00 – 0xFF	red 0x00 - 0xFF

Type Definitions

Representing Colors

- We can use C *bit-wise operators* to build and extract the RGBA fields. Here are examples for unsigned 8-bit values.

bits << count	shift bits left by count (fills with 0's from right)	$1\ 0\ 0\ 0\ \underline{1\ 0\ 0\ 1}$ << 4 $= \underline{1\ 0\ 0\ 1}\ 0\ 0\ 0\ 0$ ← left 4
bits >> count	shift bits right by count (fills with 0s from left)	$\underline{1\ 0\ 0\ 1}\ 1\ 0\ 0\ 1$ >> 4 $= 0\ 0\ 0\ 0\ \underline{1\ 0\ 0\ 1}$ → right 4

bits1 & bit2 — bits in bits1 that are also in bits2 (bits2 *masks* bits1)

	$\underline{1\ 0\ 1\ 0}\ \underline{1\ 0\ 1\ 0}$	bits1 with two 4-bit fields
&	$0\ 0\ 0\ 0\ \underline{1\ 1\ 1\ 1}$	bits2 masks lower 4 bit field in bits1
=	$0\ 0\ 0\ 0\ \underline{1\ 0\ 1\ 0}$	only lower 4 bits of bits1 retained

bits1 \| bit2 — bits that are in either bits1 *or* bits2

	$\underline{1\ 0\ 1\ 0}\ 0\ 0\ 0\ 0$	upper 4-bit field in bits1
|	$0\ 0\ 0\ 0\ \underline{1\ 0\ 1\ 0}$	lower 4-bits field in bits2
=	$\underline{1\ 0\ 1\ 0}\ \underline{1\ 0\ 1\ 0}$	combines upper, lower 4-bit fields

Type Definitions

Representing Colors

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>

/** Color representation for components little-endian a b g r */
typedef  uint32_t  Color;

/**
 * Create a color from the components of another color
 * @param c the color to set
 * @return a copy of the color
 */
Color makeColorFromColor(Color c) {
    return c;   // returns uint32_t field by value
}
```

Type Definitions

Representing Colors

```
/**
 * Create a color from r, g, b, and a components
 * @param r red color component
 * @param g green color component
 * @param b blue color component
 * @param a alpha color component
 * @return color
 */
Color makeColorFromRGBA(uint8_t r, uint8_t g, uint8_t b, uint8_t a) {
    Color c = a;             // set alpha field
    c = (c << 8) | b;        // shift fields up and add b field
    c = (c << 8) | g;        // shift fields up and add g field
    c = (c << 8) | r;        // shift fields up and add r field
    return c;
}
```

Type Definitions

Representing Colors

```
/**
 * Determine whether two colors are equal.
 * @param color1 the first color
 * @param color2 the second color
 * @return true if the two colors are equal
 */
bool equalsColor(const Color color1, const Color color2) {
    return color1 == color2;  // == operator on Color (uint32_t)
}
```

Type Definitions

Representing Colors

```
/**
 * Get red color component.
 * @param color the color
 * @return red color component
 */
uint8_t getRedOfColor(Color c) {
	return c & 0xFF;     // extract r field
}

/**
 * Get green color component.
 * @param color the color
 * @return green color component
 */
uint8_t getGreenOfColor(const Color c) {
	return (c >> 8) & 0xFF;   // shift fields down 8 bits and extract g field
}
```

Type Definitions

Representing Colors

```
/**
 * Get blue color component.
 * @param color the color
 * @return blue color component
 */
uint8_t getBlueOfColor(const Color c) {
	return (c >> 16) & 0xFF;	// shift fields down 16 bits and extract g field

}

/**
 * Get alpha color component.
 * @param color the color
 * @return alpha color component
 */
uint8_t getAlphaOfColor(const Color c) {
	return (c >> 24) & 0xFF;	// shift fields down 16 bits and extract a field
}
```

Type Definitions

Defining a Boolean Type

- Here is a main function to test our new Color data type.

```
/** Test Color data type and functions */
int main(void) {
    Color c1 = makeColorFromRGBA(0xFF, 0xF0, 0x0F, 0x01);
    uint8_t r1 = getRedOfColor(c1);
    uint8_t g1 = getGreenOfColor(c1);
    uint8_t b1 = getBlueOfColor(c1);
    uint8_t a1 = getAlphaOfColor(c1);
    printf( “Color c1:”
            ” r: 0x%02x"
            " g: 0x%02x"
            " b: 0x%02x"
            " a: 0x%02x\n", r1, g1, b1, a1);
```

Type Definitions

Defining a Boolean Type

- Here is a main function to test our new Color data type.

```
    Color c2 = makeColorFromColor(c1);
    uint8_t r12= getRedOfColor(c2);
    uint8_t g2 = getGreenOfColor(c2);
    uint8_t b2 = getBlueOfColor(c2);
    uint8_t a2 = getAlphaOfColor(c2);
    printf( “Color c2:”
            ” r: 0x%02x"
            " g: 0x%02x"
            " b: 0x%02x"
            " a: 0x%02x\n", r2, g2, b2, a2);

    // determine whether c1 and c2 are equal
    bool eq = equalsColor(c1,c2);
    printf("Color c1 == c2? %s", eq ? "true" : "false");
}
```

Type Definitions

Representing Colors

- This representation has both advantages and disadvantages.
- Advantages:
  - A Color can be created and returned within the makeColor() function.
  - A Color can easily be copied, assigned, or compared to another color in the same way as a basic C type.
- Disadvantages:
  - The representation is not as straight-forward or easy to work with.

Type Definitions

Representing Colors

- Both representations share the advantage that working with Color does not depend on which typedef is used.
- A Color is created by functions that initializes its r, g, b, a appropriately, and accesses its values appropriately for the typedef.
- Color is called an *abstract data type* and its functions are said to *encapsulate* its typedef representation.
- These are important concepts that we will build on in this and future courses.

Type Definitions

Typedefs and Functions

- We saw earlier how to use pointers to scalar and array values. Now we will learn about pointers to functions.
- Here is a function that compares two ints and returns true if the first is greater than the second.

```
/**
 * Compares two int values for greater than.
 * @param val1 the first value
 * @param val2 the second value
 * @return true if val1 is greater, false otherwise
 */
bool compareGreater(int val1, int val2) {
    return val1 > val2;
}
```

Type Definitions

Typedefs and Functions

- In C, a function name is actually a const pointer to a function definition.
- Here is the declaration of a function pointer myFunct that also points to the compareGreater() function:

```
bool (*myFunct)(int, int) = compareGreater;
```

- Given a similar function compareLess(), we could also make myFunct point to that function instead:

```
myFunct = compareLess;
```

- Whichever function that myFunct points to can be called as

```
bool result = myFunct(3, 5);
```

Type Definitions

Typedefs and Functions

- We can create a typedef for the type of these comparison functions with two int parameters and return a bool.

  ```
  typedef bool (*CompareFunct)(int, int);
  ```

- This enables us to declare a pointer to a comparison function using the typedef rather than spelling out the type for each declaration.

  ```
  CompareFunct myFunct = compareGreater;
  ```

- It also makes it more convenient to pass a pointer to a comparison function to another function that uses it to perform whatever comparison is specified.

Type Definitions

Typedefs and Functions

```
/**
 * Return the value from array selected by compare function.
 * @param n the array size
 * @param array the array
 * @param compare the compare function
 * @return the selected value from the array
 */
int select(size_t n, int array[n], CompareFunct compare) {
    int val = array[0];  // initially select first element
    for (int i = 1; i < n; i++) {
        if (compare(array[i], val)) {
            val = array[i];        // select this element if compares
        }
    }
    return val;
}
```

Type Definitions

Typedefs and Functions

```
/** Test the comparison functions */
int main(void) {
    // test array for selection
    int array[] = {-7, 4, 7, 3, 2, -3};

    // select the least element in the array
    int ltVal = select(6, array, compareLess);
    printf("Least value in array is %d\n", ltVal);

    // select the greatest element in the array
    int gtVal = select(6, array, compareGreater);
    printf("Greatest value in array is %d\n", gtVal);
}
```

Output:

```
Least value in array is -7
Greatest absolute value in array is 7
```