2.30 advanced types

Defining new type aliases (typedef)

Enumerations (enum)

Structures (struct)

EE108 – Computing for Engineers

1

Overview

Aims

- Learn how to use enum, struct, and typedef

Learning outcomes – you should be able to…

- Define enumerated types and use enumerated type variables and function parameters
- Define structure types and use structure type variables and function parameters
- Use both the dot notation and arrow notation for accessing structure members
- Use typedef to create simpler or more readable aliases for existing types (particularly commonly used with enum, struct, and union types)

2

typedef

3

28 November 2020

3

Type definition

4

We can define new types, or more accurately, new type names that the compiler will recognise using the **typedef** keyword

- The new type name is an alias for an existing type
- The new type name can then be used just like the existing type in variable/const declarations, in function parameter and return type definitions/declarations, with the cast operator, etc.

The syntax is

```
typedef  existingType  newTypeName;
```

```
E.g.  typedef   unsigned char   Byte;
```

A typedef’d alias can be used wherever the existing type could have been used (see examples on next slide)

28 November 2020

4

Basic typedef examples

```
// define a new alias, Uint32, for an unsigned int that is 4 byte (32 bits) wide.
// On Arduino an unsigned long is 32 bits wide, so we can define Uint32 as
// an alias for unsigned long. (On other platforms, e.g. the PC, a Uint32 might
// correspond to a unsigned int. However we write the rest of our code to use
// Uint32 so that we only need to change one line to adapt to new platforms.)
// Note: that the our naming convention for type aliases is to use mixed
// case and start with a capital letter and not a lower case letter
typedef   unsigned long   Uint32;

// define a new alias which we'll use when printing pointer values
typedef   unsigned int    PrintablePointer;

// compare the use of the existing type and typedef'd alias in the following
unsigned long x1 = 1;
Uint32        x2 = 2;
unsigned long *p1 = &x1; // pointer to x1
Uint32        *p2 = &x2; // pointer to x2
Uint32        *p3 = &x1; // legal since unsigned long and Uint32 are identical

Serial.print("Pointer values (i.e. addresses) are ");
Serial.println((unsigned int) p1); // force pointer to print as a number
Serial.println((PrintablePointer) p2); // force pointer to print as a number with
                                       // a more self-explanatory cast
```

Self test questions

Q1. Define a new type, Milliseconds, which should be an alias for the type returned by the function millis(), i.e. an unsigned long.

Q2. Declare a variable, startTime, of the type you just defined and initialize it to return value from the millis() function.

Q3. Define a new type, Uint8, which should be an alias for an 8-bit wide unsigned integer.

Q4. Declare a variable, foo, of the type you just defined and initialize it to 27.

Self test questions

Q5. Define a new type, IntPtr, which should be an alias for a pointer to an int type.

Q6. Declare a pointer variable, myPtr, of the type you just defined and initialize it to the NULL pointer.

Self test questions

In C a variable that will refer to a literal or read-only string might be defined as follows:

```
const char * myString = “hello”;
```

Q7. Define a new type, CString, which can be used to simplify the declaration of C strings such as the example given above.

Q8. Declare a variable, myString, of the type you just defined in Q7 and initialize it to the literal string “hello”. Ensure that the declaration is valid and will be considered identical to the initial example by the compiler.

Enumerations

10

28 November 2020

10

Enumeration Types

11

An enumeration is a data type used to define a set of related constants, e.g.

```
// define a new enumerated type to represent a card suit
// The enum tag is “Suit” in this case
enum Suit { SUIT_DIAMONDS, SUIT_HEARTS, SUIT_CLUBS, SUIT_SPADES };
```

Enum tag

An enumeration variable is declared and used as follows

```
…
enum Suit   suit1;  // declaration of variable suit1, no value defined
enum Suit   suit2 = SUIT_DIAMONDS; // declaration of suit2 with initializer

suit1 = SUIT_HEARTS; // assign to suit1
```

The main motivation for enumerations is

- to make the relationship between certain constants explicit
- to suggest that, in a given context, only a restricted set of values is valid

28 November 2020

11

Enumeration scope

Enumerated types have the same scope as variables or consts

The constant names in an enumerated type must be unique within the scope that the enumeration is declared in. It is not possible for the same constant name to appear in two different enumerated types in the same scope.

```
// the following won't compile because the constant name SIZE
// is not unique in the scope
enum Day { MON=0, TUE, WED, THU, FRI, SAT, SUN, SIZE };
enum Season { SPRING, SUMMER, AUTUMN, WINTER, SIZE };
```

The solution is to name the constants within each enum uniquely.

Name enum constants by converting the enum tag (or an abbreviation of it) to UPPERCASE and use this as a prefix on each of the constants

```
// this is a solution to the problem shown above
enum Day {
  DAY_MON=0, DAY_TUE, DAY_WED, DAY_THU, DAY_FRI, DAY_SAT, DAY_SUN, DAY_SIZE
};
enum Season {
  SEASON_SPRING, SEASON_SUMMER, SEASON_AUTUMN, WINTER, SEASON_SIZE
};
```

Enumerations as integers

```
enum Suit { SUIT_DIAMONDS, SUIT_HEARTS, SUIT_CLUBS, SUIT_SPADES };
…
enum Suit suit1;
```

In C, enumerations are treated almost identically to constant integers

- You can use enumeration variables (e.g. suit1) and enumeration constants (e.g. SUIT_HEARTS) wherever you can use an int (e.g. as an index into an array)

If you don't specify it explicitly

- the first constant in an enumeration has a default value of zero
- Consecutive constants in the enumeration, have consecutive values
- E.g. in the code at top of slide, SUIT_DIAMONDS defaults to zero, SUIT_HEARTS to 1, SUIT_CLUBS to 2, etc.

Enumeration constant values

14

You can explicitly specify the values of enumeration constants

- You can specify any integer value you like and the order is not important
- It is legal for more than one enumeration constant to have the same value

```
enum Delay {
  DELAY_MIN = 50,
  DELAY_MAX = 500,
  DELAY_MEDIUM = 200,  // note that order doesn’t matter here
  DELAY_X,             // value of DELAY_X will be 201 (1 more than
                       // previous constant)
  DELAY_DEFAULT = 200  // note it is allowed to have same value as
                       // another constant in the enumeration
};
```

28 November 2020

14

Enums and tables/arrays

15

It is common in C to use enums to specify different values in a table/array

In this case

- set the value of the first constant explicitly
- Allow all other enum constants to take on their default value (i.e. previous constant plus 1)
- Add a constant at the very end whose name ends with _END, _SIZE, _LEN, or similar

E.g.

```
enum Suit {
  SUIT_DIAMONDS = 0, // specify the first value is zero
  SUIT_HEARTS,       // allow subsequent constants to take on their
  SUIT_CLUBS,        // default value
  SUIT_SPADES,
  SUIT_SIZE           // special element used to mark end of constants
};
```

28 November 2020

15

Contd.

16

The convention on the previous slide makes it easier to use enums with arrays, e.g.

```
enum Suit {
  SUIT_DIAMONDS = 0, // specify the first value is zero
  SUIT_HEARTS,       // allow subsequent constants to take on their
  SUIT_CLUBS,        // default value
  SUIT_SPADES,
  SUIT_SIZE           // special element used to mark end of constants
};

…

// The following array can be used to map int constant values to printable
// strings. It is critical to make sure the order of strings matches the order
// of constants in the corresponding enum.
// Note: SUIT_SIZE is automatically the correct size for the “valid” SUIT
// entries because it was placed as the last enum entry above
const char *SUIT_NAMES[SUIT_SIZE] = { “diamonds”, “hearts”, “clubs”, “spades” };
int i;

for (i=SUIT_DIAMONDS; i<SUIT_SIZE; i++)
  Serial.println(SUIT_NAMES[i]);
```

28 November 2020

Typedef and enum

17

You can use typedef to save having to write enum every time you declare a variable

```
// EXAMPLE WITHOUT TYPEDEF
// first declare the enum type
enum Suit { SUIT_DIAMONDS, SUIT_HEARTS, SUIT_CLUBS, SUIT_SPADES };
…
// later to declare a variable or parameter with the enum type we have to write
// enum and the enum tag name (Suit in this case)
enum Suit suit1;
enum Suit suit2 = SUIT_CLUBS;
```

```
// EXAMPLE WITH TYPEDEF (long version)
// first declare the enum type
enum Suit { SUIT_DIAMONDS, SUIT_HEARTS, SUIT_CLUBS, SUIT_SPADES };
// then separately typedef the alias for the enum type
typedef  enum Suit  Suit; // note “enum Suit” is existing type
                          // while “Suit” on its own is the new alias
…
// now declare variables using the alias rather than the original enum type
Suit suit1;
Suit suit2 = SUIT_CLUBS;
enum Suit suit3 = SUIT_HEARTS; // you can still use original enum type
```

28 November 2020

Typedef and enum contd.

```
// EXAMPLE WITH TYPEDEF (shorter version, but still including enum tag)

// rather than declare the enum first and typedef afterwards, it is possible
// to combine these two statements in one line
typedef enum Suit { SUIT_DIAMONDS, SUIT_HEARTS, SUIT_CLUBS, SUIT_SPADES }  Suit;
…
// now declare variables using the alias rather than the original enum type
Suit suit1;
Suit suit2 = SUIT_CLUBS;
enum Suit suit3 = SUIT_HEARTS; // you can still use original enum type
```

```
// EXAMPLE WITH TYPEDEF (shorter version, but with anonymous enum)

// As in example above, but here the existing type is an anonymous enum
// (it has no enum tag) and Suit is the new alias. Because enum has no tag
// you cannot use it directly – you must use the typedef'd alias
typedef enum { SUIT_DIAMONDS, SUIT_HEARTS, SUIT_CLUBS, SUIT_SPADES }  Suit;
…
// now declare variables using the alias rather than the original enum type
Suit suit1;
Suit suit2 = SUIT_CLUBS;
// enum ??? suit3 = SUIT_HEARTS; // you cannot use the original enum type
                                 // because there is no tag to identify it
```

Self test questions

Q1. A soccer robot can execute diving saves or sit-down saves to the left or the right. Declare an enumerated type to represent these possibilities with constants.

Q2. Complete the following partial code of the loop function based on the comments.

```
void loop() {
  …
  // check if ball will intersect goal line on far left
  if (ballGoalIntersection.y > 0.5) {
    // call goalieSave function with the appropriate save type from Q1



  }
  …
}
```

Self test questions

Q3. Fill in the missing pieces of the goalieSave function definition (to be called from Q2) based on the following .

```
void goalieSave( ___________________________________  saveType) {
  …
  if (saveType ==  ___________________________________ ) {
     executeDivingSaveLeft();
  } else if …
    …
}
```

Q4. Define a typedef'd alias for the enum in Q1. How would this affect your answer to Q3?

Structures

Structures

In C, arrays are used to group or store several data items of the same type, without giving them any special symbolic names

- For example an array of int just stores elements of type int
- A 2d Cartesian coordinate could be represented using an array of two elements, but you as programmer, would have to remember that element 0 is the x-coordinate, whereas element 1 was the y-coordinate

Structures are used to group or store together several data items that may have different types, and to give symbolic names to each of the data items

- E.g. A 2d Cartesian coordinate could be represented by two int data items refered to as x and y
- A book in a library might be identified by a record with data items for the book title (a string) and book number (a long long), etc.

The grouped data items in a C structure are referred to as the structure members (or structure fields)

Structure Type

A structure is a data type used to define a set of related data members which should always be grouped and stored together.

E.g. In Linux, the following standard structure defines an accurate time

Structure tag

```
struct timespec { // the structure tag is timespec
  int    tv_sec;  // seconds – (slight simplification of the actual structure)
  long   tv_nsec; // nanoseconds
};
```

To declare a variable of this structure type we can declare it with or without initializer:

```
// declaration of variable startTime, with *no initializer*. The values of
// the tv_sec and tv_nsec members of startTime are undefined
struct timespec   startTime;

// declaration of timeout variable with an initializer. The value of
// tv_sec is 0 and tv_nsec is 10,000,000 i.e. 10 ms.
// (Compare the appearance of a structure initializer to an array
// initializer.) Values get assigned to structure fields in the order
// they were declared
struct timespec   timeout = { 0, 10000000 } ;
```

Another example

Note that structure members can have different types to each other (e.g. float, int, char *, enum, etc.)

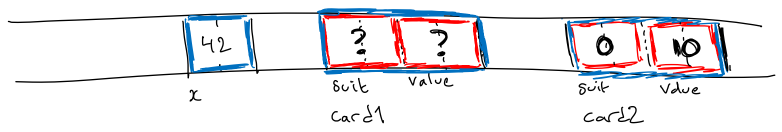
E.g.

```
enum Suit { SUIT_DIAMONDS = 0, SUIT_HEARTS, SUIT_CLUBS, SUIT_SPADES };

// define a new structure to represent information about a playing card
struct Card { // the structure tag is Card
  enum Suit  suit;
  int        value;
};
```

Declarations with/without initializer for the Card structure.

```
// declaration of variable card1, with no initializer. The values of
// the suit and value members of card1 are undefined
struct Card   card1;

// declaration of card2 with a structure initializer. The value of suit
// is SUIT_DIAMONDS and value is 10. (Compare the appearance of a
// structure initializer to an array initializer)
struct Card   card2 = { SUIT_DIAMONDS, 10 } ;
```

Visualizing the Card structure in memory



int x = 42;

struct Card card1;

struct Card card2 = { SUIT_DIAMONDS, 10 };

Using a structure variable – member access

To use a structure variable you will generally want to get or set the value of its members

If we have access to the structure variable, we can access its members using the **dot notation**

```
structureVariable.memberName
```

```
…
struct Card card1;
int tmp;

// set (write) the value of a structure's members
card1.suit = SUIT_HEARTS;
card1.value = 5;

// get (read) the value of a structure's members
tmp = card1.value;
if (card1.suit == SUIT_CLUBS)
  …
```

Self test questions

Q1. Declare a structure type to represent a fraction with two whole numbers.

Q2. Declare an uninitialized variable of the type declared in Q1.

Q3. Declare a variable of the type declared in Q1 and initialize it to represent 22/7

Self test questions

Q4. Consider the following definitions and declarations

```
struct Location {
  int x;
  int y;
  float confidence;
};

…

struct Location ballLocation = { 0, 1, 0.5 };
```

Write code to check if the sum of the ballLocation x and y members is less than 3 and if it is, set the confidence member to 0.7.

Typedef and struct

You can use typedef to save having to write struct *tagname* every time you declare a structure variable

```
// WITHOUT TYPEDEF
// first declare the struct type
struct Date {
  int day;
  int month;
  int year;
};
…


// later to declare a variable
// or parameter we must use
// struct tagname, e.g.
struct Date date1;
```

```
// WITH TYPEDEF – longer version
// first declare the struct type
struct Date {
  int day;
  int month;
  int year;
};
// then declare a typedef’d alias
typedef   struct Date    Date;

// later, to declare a variable
// just use the typedef’d alias, e.g.
Date date1;

// we can also use the original struct
// type instead of the alias if we want
struct Date date2;

// struct Date is the existing type
// Date (on its own) is the alias
```

Typedef and struct contd.

31

You can use typedef to save having to write struct *tagname* every time you declare a structure variable

```
// WITH TYPEDEF – shorter version, but
// still declare a structure tag

typedef struct Date {
  int day;
  int month;
  int year;
} Date; // Date is the new alias for
        // struct Date
…
// later, to declare a variable
// just use the typedef'd alias, e.g.
Date date1;

// we can also use the original struct
// type instead of the alias if we want
struct Date date2;

// struct Date is the existing type
// Date (on its own) is the alias
```

```
// WITH TYPEDEF – shorter version, but
// eliminate the structure tag

typedef struct { // anonymous struct
  int day;
  int month;
  int year;
} Date; // Date is the new alias for
        // the anonymous struct
…
// later, to declare a variable
// just use the typedef'd alias, e.g.
Date date1;

// we cannot use the original struct
// type because it has no tag
// struct ??? date2; // illegal
```

28 November 2020

Self test questions

32

Consider the following definitions and declarations

```
struct Location {
  int x;
  int y;
  float confidence;
};
```

Q1. Use typedef to declare Location as an alias for struct Location (without modifying the above) and declare a variable, foo, of this new typedef'd type.

28 November 2020

Self test questions

33

Q2. Use typedef to declare Location as an alias for struct Location but write the declaration in one line. (Shorter form but still including struct tag)

Q3. Use typedef to declare Location as an alias for struct Location but write the declaration on one line using an anonymous struct (shorter form with no struct tag)

28 November 2020

33

Nested structures

34

It is possible to define **nested structure** type, where the type of (some) of the outer structure members are themselves structures

```
struct Date {
  int day;
  int month;
  int year;
};

// declaring a structure with
// members that are themselves
// structures
struct Duration {
  struct Date start;
  struct Date end;
}

// declare nested structure with
// initializer
struct Duration dur1 = {
  { 1, 2, 2013 },  // start values
  { 12, 4, 2013 }  // end values
};
```

```
struct Duration dur2;
int tmp;
…

// set the value of a nested
// structure’s members
dur2.start.day = 1;
dur2.start.month = 2;
dur2.start.year = 2013;
dur2.end.day = 12;
dur2.end.day = 4;
dur2.end.day = 2013;

// get the value of a nested
// structure’s members
tmp = dur1.start.month;
if (dur1.end.year == dur1.start.year)
  …
```

28 November 2020

34

Arrays of structures

It is also possible to define arrays of structures, where the array element type is a structure

```
struct Date {
  int day;
  int month;
  int year;
};

// declare array of structures
// *with* initializer
struct Date labDates[] = {
  { 1, 2, 2013 }, // init Date 0
  { 15, 2, 2013 }, // init Date 1
  { 22, 2, 2013 } // init Date 2
};

// declare array of structure
// *without* initializer -- element
// values will be undefined
struct Date dueDates[3];
```

```
// set the value of element members
// in a structure array
dueDates[2].day = 1;
dueDates[2].month = 2;
dueDates[2].year = 2013;

// get the value of element members
// in a structure array
tmp = labDates[2].month;
if (labDates[1].year == labDates[0].year)
  …
```

Self test questions

Q1. Declare a structure called TimeSpan which has 2 members, startTime and endTime and both of these members are of type struct timespec (see slide 24)

Q2. Declare an uninitialized array variable called timespans of size 2 whose elements are TimeSpan structures as defined in Q1.

Self test questions

37

Q3. Initialize the members of element 0 of the timespans array such that the start time is 0 secs, 0 nanosec, and the end time is 1 sec, 0 nanosec.

28 November 2020

37

Comparing and assigning structures

38

You can assign structure variables (in addition to assigning individual members),

You <u>cannot directly compare</u> structure variables in C

- Instead you must perform comparison of corresponding members individually

```
struct Date { int day; int month; int year; };
struct Date date1 = { 1, 1, 2013 };
struct Date date2;

date2 = date1; // all values of date1 member are copied to date2 members

if (date2 == date1) // illegal – can’t compare structures directly
  …

// the following is the legal way of comparing structures – you must
// do it member by member
if ((date2.day == date1.day)
       && (date2.month == date1.month)
       && (date2.year == date1.year))
  …
```

28 November 2020

38

Structures and functions

Functions may be defined to have parameters or return types that are structures – but usually you should almost always use structure pointers instead (see later slide)

⚠ ***Warning:** Although using structure parameters as below is legal, it results in inefficient code since the value of all members of any structure arguments or structure return values must be copied. In general it is not recommended.*

```
struct Date earliestByYear(struct Date date1, struct Date date2) {
  if (date1.year < date2.year)
    return date1;
  else
    return date2;
}

…
struct Date firstDate = { 1, 2, 2013 };
struct Date secondDate = { 2, 3, 2013 };
struct Date anotherDate;

anotherDate = earliestByYear(firstDate, secondDate);
```

Structure pointers

We use structure pointers for much the same reason we use scalar pointers

Structure pointers allow us to write functions that only need to know the structure type and not the specific structure variables

In addition, it is inefficient to pass structures by value since every member of the structure must be copied. It is much better to pass by reference (i.e. as a pointer)

Structure pointers

Pointers to structure variables are declared just like pointers to other variable types such as int and float

```
struct Date { int day; int month; int year; };
struct Date date1; // ordinary structure variable
struct Date *pDate; // structure pointer, currently uninitialized

pDate = &date1; // pDate now points to date1
```

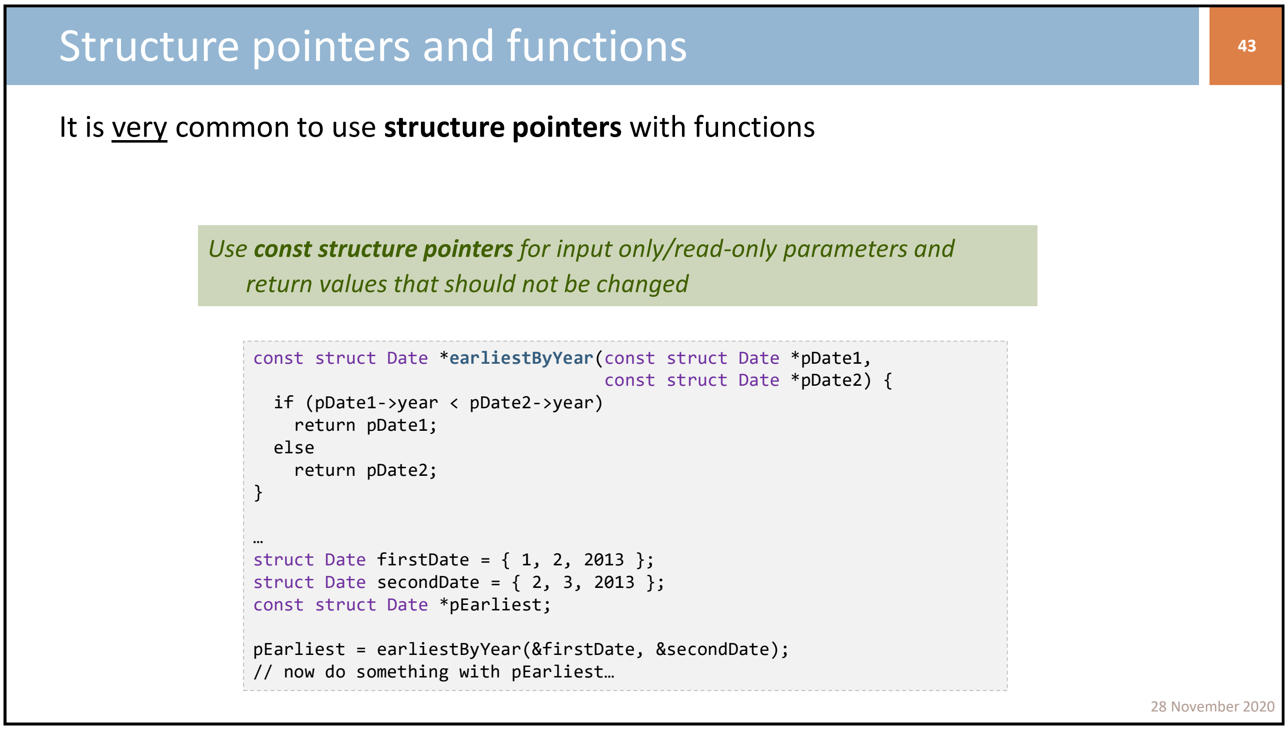
To set/get members of a structure pointee via a structure pointer we use the **arrow notation**

```
int tmp;

// pDate->day selects the member day in pDate’s pointee.
// Here we get its value
tmp = pDate->day;
// pDate->month selects the member month in pDate’s pointee.
// Here we set its value
pDate->month = 12;
```

Contd.

The arrow notation is shorthand for the dereference operator (*) and the member selection operator (.)

```
// continuing example from previous slide…

// pDate->day selects the member, day, in what’s pointed at by pDate
tmp = pDate->day;

// equivalent using the normal pointer notation would be as follows.
// Parentheses are required to indicate that we want to dereference
// pDate and not pDate.day (which would be invalid since pDate is not
// a structure and even if it was, the member day is not a pointer)
tmp = (*pDate).day;

// pDate->month selects the member, month, in what’s pointed at by pDate
pDate->month = 12;

// the equivalent using normal pointer notation
(*pDate).month = 12;
```



Structure pointers and functions

43

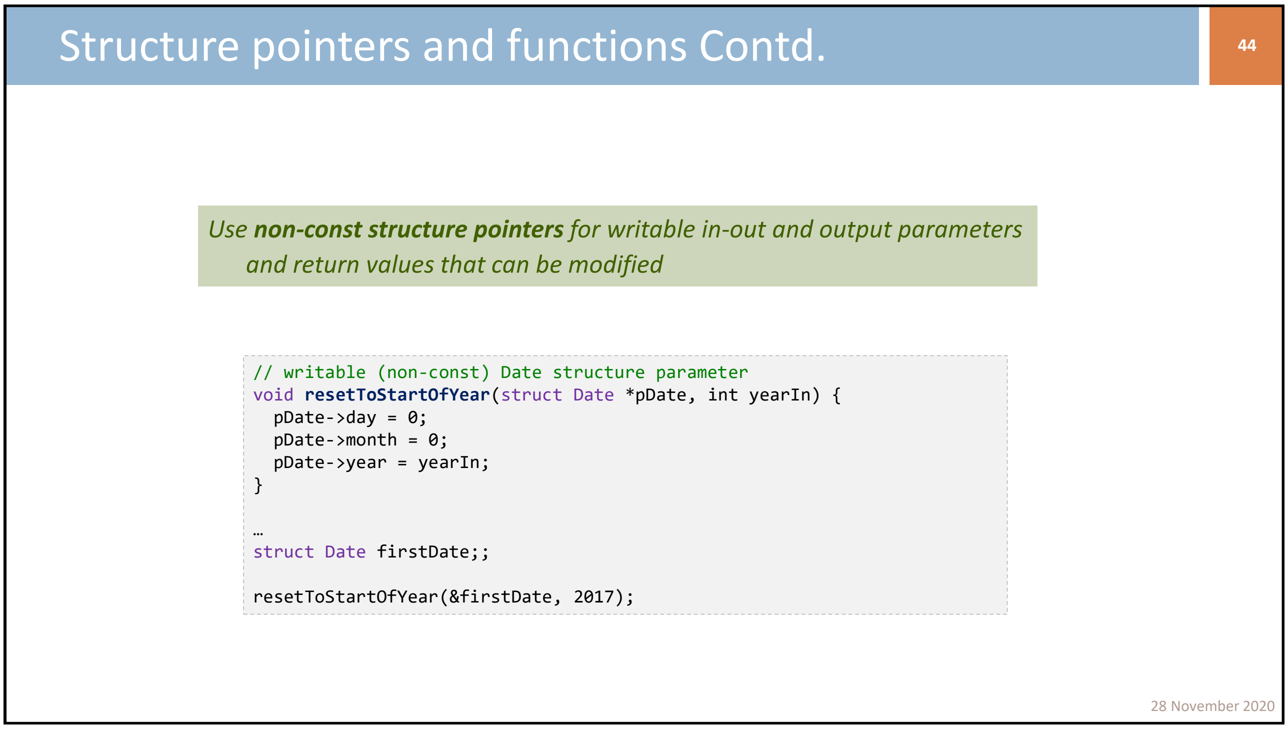
It is <u>very</u> common to use **structure pointers** with functions

*Use **const structure pointers** for input only/read-only parameters and return values that should not be changed*

```
const struct Date *earliestByYear(const struct Date *pDate1,
                                  const struct Date *pDate2) {
  if (pDate1->year < pDate2->year)
    return pDate1;
  else
    return pDate2;
}

…
struct Date firstDate = { 1, 2, 2013 };
struct Date secondDate = { 2, 3, 2013 };
const struct Date *pEarliest;

pEarliest = earliestByYear(&firstDate, &secondDate);
// now do something with pEarliest…
```

28 November 2020

43



Structure pointers and functions Contd.

44

*Use **non-const structure pointers** for writable in-out and output parameters and return values that can be modified*

```
// writable (non-const) Date structure parameter
void resetToStartOfYear(struct Date *pDate, int yearIn) {
  pDate->day = 0;
  pDate->month = 0;
  pDate->year = yearIn;
}

…
struct Date firstDate;;

resetToStartOfYear(&firstDate, 2017);
```

28 November 2020

44

Self test questions

Q1. Define a function called resetLocation which resets the members of a Location structure (see slide 29) back to 0,0 for x and y and 1.0 for confidence.

Q2. Declare a Location structure variable called ***home*** *(leave it uninitialized) and then call resetLocation (that you defined in Q1) to reset its member values.*

Self test questions

Q3. Define a function called checkLocation which checks that both the x and y members of a Location structure are less than 2 and that the confidence is greater than 0.5. If all these conditions are met, return true, otherwise return false.

Q4. Using the variable ***home*** *declared in Q2 (previous slide) call checkLocation (that you defined in Q3) to check the location and to print "location good" if checkLocation returns true.*