CMPT 125: Introduction to Computing Science and Programming II

Spring 2023

Week 3: Composite data types, functions, memory allocation

Instructor: Victor Cheung, PhD

School of Computing Science, Simon Fraser University

Joke of the day

THERE'S BEEN A LOT OF CONFUSION OVER 1024 vs 1000, KBYTE vs KBIT, AND THE CAPITALIZATION FOR EACH.

HERE, AT LAST, IS A SINGLE, DEFINITIVE STANDARD:

SYMBOL	NAME	SIZE	NOTES
kB	KILOBYTE	1024 BYTES OR 1000 BYTES	1000 BYTES DURING LEAP YEARS, 1024 OTHERWISE
KB	KELLY-BOOTLE STANDARD UNIT	1012 BYTES	COMPROMISE BETWEEN 1000 AND 1024 BYTES
KiB	IMAGINARY KILOBYTE	1024 √-1 BYTES	USED IN QUANTUM COMPUTING
kb	INTEL KILOBYTE	1023.937528 BYTES	CALCULATED ON PENTIUM F.P.U.
Kb	DRIVEMAKER'S KILOBYTE	CURRENTLY 908 BYTES	SHRINKS BY 4 BYTES EACH YEAR FOR MARKETING REASONS
KBa	BAKER'S KILOBYTE	1152 BYTES	9 BITS TO THE BYTE SINCE YOU'RE SUCH A GOOD CUSTOMER

https://xkcd.com/394/

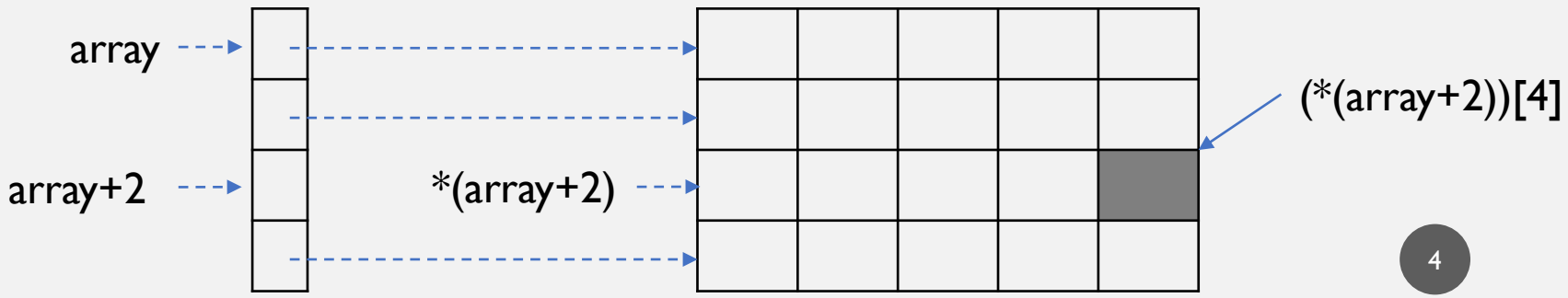
Recap from Last Lecture

- Strings and string functions
  - Immutable strings (3 main ways to create a C string)
  - When a string is not found inside another string, strstr returns a null pointer
- Reading user input
  - The scanf function, similar syntax as printf, except direction is different and is pass-by-reference
- 2D arrays
  - Can be thought of as an array of pointers to arrays

Review from Last Lecture (1)

- Investigate if this statement is valid: char* str5 = {'w', 'o', 'r', 'd', '\0'};
  - It is not valid. This is because the expression **char*** is considered as the type **pointer to a character**, not a char array
  - The expression **char* str3 = "word"** causes the compiler to behave differently with the string literal

- Another way to access a 2D array is to consider the fact that it is simply a pointer to an array of pointers, investigate what this expression is accessing in a 4-by-5 2D array (e.g., int array[4][5]):
  - (*(array+2))[4]



Review from Last Lecture (2)

- Like 1D arrays, 2D arrays can also be passed to a function as parameters
  - The function must first know the number of columns so it can calculate internally where the next row is in the memory

```
//note the order of parameters, must be this way
void addOne(unsigned int rows, unsigned int columns, int array[][columns]) {
    for (int r=0; r<rows; r++) {
        for (int c=0; c<columns; c++) {
            array[r][c] += 1;
        }
    }
}
```

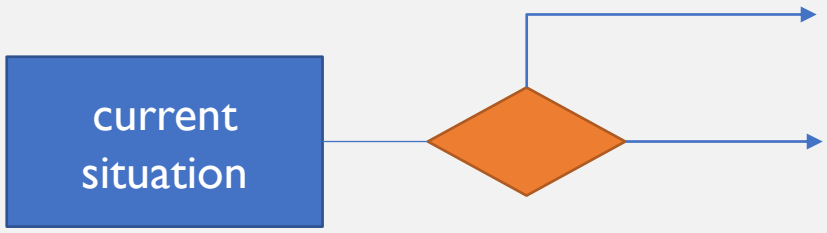
```
int int2Darray[2][3] = {{1, 3, 5}, {2, 4, 6}};
int int2Darray2[3][4] = {{1, 1, 1, 1}, {2, 2, 2, 2}, {3, 3, 3, 3}};
addOne(2, 3, int2Darray);
addOne(3, 4, int2Darray2);
```

Today

- Conditionals
  - if-statements
  - switch-statements
- Composite data types (struct & enum)
- Functions

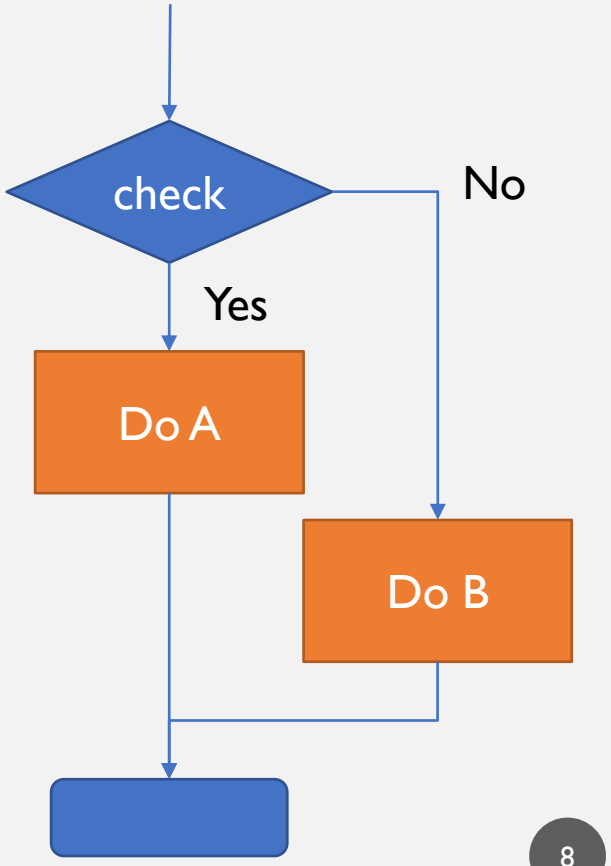
Intelligent Programs

- To be intelligent, on top of interactive, a program needs some way to **make decisions**
- This means the program needs to be able to check the current situation (state) of the program
  - the "current situation (state)" is called **condition** and typically is represented by some values stored in the variables



Conditionals

- Conditionals are programming structures that **control the flow of the program** by determining which code is executed and which code is not, based on a condition
- A condition is an expression that yields a yes/no answer, for example
  - *are two values equal? is one value larger than the other?*
    is a string inside another? are two strings a match?
  - can be a combination of expressions: *some values equal AND some larger*
    - In C/C++, the logical operator of AND is &&, OR is ||, NOT is !



The Boolean Variable Type

- Conditions in C are represented by a different variable type called Boolean that has 2 possible values: true/false
  - Allows you to use a variable as a "**flag**" to indicate something being yes/no, on/off, …etc.

Need to include the stdbool.h library

Loop ends when an even number is found or j goes beyond the array

```
int j = 0;
bool evenNumberFound = false; //start with not found
int array[10] = {-1, 0, 2, -2, 4, 10, 6, 7, 2, 9};
while (!evenNumberFound && j<10) {
    if (array[j] % 2 == 0) {//remainder is 0, so it's even
        evenNumberFound = true;
    }
    j++; //move on to look at the next number
}

if (evenNumberFound) {
    printf("There is at least 1 even number in the array.\n");
} else {
    printf("There is not any even number in the array.\n");
}
```

If-Statements

- 3 kinds of if-statements
  - If there is only one statement for the action, the { } can be omitted. But as a habit, use them anyway

```
if (...) {
    //do A
}
```

A is either executed or not

```
if (...) {
    //do A
} else {
    //do B
}
```

Only one of A & B is executed

```
if (...) {
    //do A
} else if (...) {
    //do B
} else {
    //do C
}
```

Only one of A, B, & C is executed

If-Statements Examples (1)

```c
int i = 3;

if (i == 3) {
    printf("i is indeed storing a value of 3.\n");
}

if (i%2 == 0) {
    printf("%d is an even number.\n", i);
} else {
    printf("%d is an odd number.\n", i);
}

if (i < 0) {
    printf("%d is a negative number.\n", i);
} else if (i > 0) {
    printf("%d is a positive number.\n", i);
} else {
    printf("%d is neither a negative nor a positive number.\n", i);
}
```

Initialize i with different numbers to see the branching

```
i is indeed storing a value of 3.
3 is an odd number.
3 is a positive number.
```

If-Statements Examples (2)

- Expressions for if-statements can be combined (&& for and, || for or), or flipped (! for not)

```
int numerator = 10, denominator = 2;
if (denominator != 0 && numerator/denominator == 5) {
    printf("%d is divisible by %d and the quotient is 5.\n", numerator, denominator);
} else { //note that in this case either (or both) of the expression is false
    printf("Either you are dividing %d by 0 or the quotient is not 5.\n", numerator);
}
```

- C performs a **shortcut** when evaluating && and ||
  - if the expression is cond1 && cond2, it will only check cond2 if cond1 is true, otherwise it'll skip cond2
  - if the expression is cond1 || cond2, it will only check cond2 if cond1 is false, otherwise it'll skip cond2

The "Closest-If" Rule

- Unlike Python, C doesn't really care about indentation
- Since the else-block is optional, indentation alone does not determine which if-block it is associated with
  - Without { } C will look for the closest-if above without an else-block for association

```
int number = 9;
if (number == 9)
    printf("The number is 9.\n");
    if (number%2 == 0)
        printf("The number is even.\n");
else
    printf("The number is not 9.\n");
```

```
The number is 9.
The number is not 9.
```

```
int number = 9;
if (number == 9) {
    printf("The number is 9.\n");
    if (number%2 == 0) {
        printf("The number is even.\n");
    }
} else {
    printf("The number is not 9.\n");
}
```

One way to avoid confusion is to use { }

Switch

- Very often we use if-else if-else to help determine an "out" based on matching, switch makes it clearer

```
int angle = 60;
switch (angle%360/90) {
    case 0: printf("First quadrant.\n");
            break;
    case 1: printf("Second quardrant.\n");
            break;
    case 2: printf("Third quadrant.\n");
            break;
    default: printf("Forth quadrant.\n");
}
```

Recall that integer/integer = integer

Each case must be a constant or literal, so C can match the values

More on The Switch Statement

- Switch will cause the control flow to jump to the first matching case
  - after that, switch will not check anymore and will keep executing the rest
    - this is called "fall through"
  - need the keyword **break** to get the control flow out of the switch body
    - this is probably the only reason you should use break in your code!
- The default case is an optional case where if nothing above matches it will be executed, similar to what **else** does in an if-statement

```
switch (marks/10) {
    case 10:
    case 9:
        puts("YOUR GRADE : A");
        break;
    case 8:
        puts("YOUR GRADE : B");
        break;
    case 7:
        puts("YOUR GRADE : C");
        break;
    default:
        puts("YOUR GRADE : Failed");
}
```

Give Switch A Break!

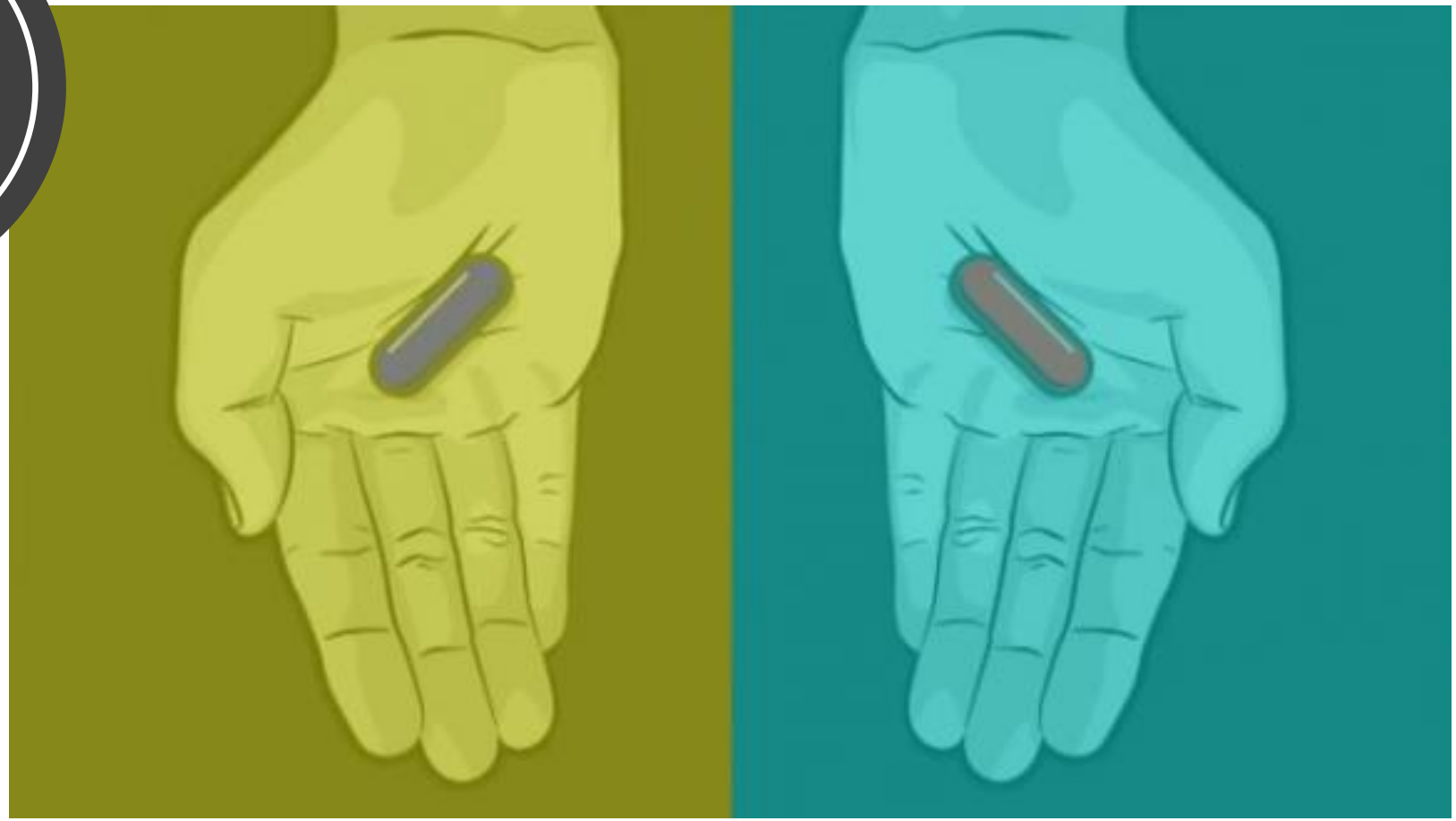
- Again, the break keyword is necessary because C will keep executing once it finds a match
  - break will make the execution jump to the first out of the switch statement

```
int angle = 60;
switch (angle%360/90) {
    case 0: printf("First quadrant.\n");

    case 1: printf("Second quardrant.\n");

    case 2: printf("Third quadrant.\n");
            break;
    default: printf("Forth quadrant.\n");
}
```

```
First quadrant.
Second quardrant.
Third quadrant.
```

BREAK FOR 10 MINUTES

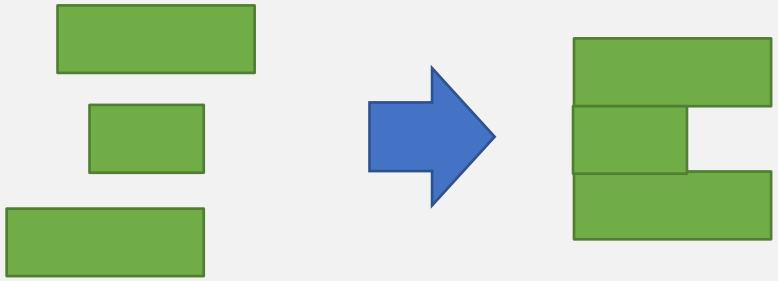
What are the colours of the pills?



Source: https://www.moillusions.com/blue-pill-red-pill-illusion

Composite Data Type

- We have learned variables that use 1 name per value, and arrays that use 1 name per group of values of the same type
- Sometimes we need more than 1 value and 1 type to store/represent an entity, for example:
  - an array of doubles and its size (so we don't need to store them separately)
  - a group of strings and numbers to represent a student (name, ID, grades, …etc.)
- C (so as many languages) provides a mechanism for us to define our own data type: Composite data type



3 separate data types

1 composite data type

Composite Data Type – Syntax (1)

- Basic syntax is to use the keyword **struct** everywhere

```
struct <compositeType> {
  <type> field1;
  <type> field2;
  <type> field3;
};

//usage
struct <compositeType> var1;
```

```
struct doubleArrayWithSize {
    unsigned int capacity;
    unsigned int used;
    double elements[10];
};
```

```
struct doubleArrayWithSize myArray;
```

Composite Data Type – Syntax (2)

- A more common way is to "typedef" it so we just need to use the type name itself

```
typedef struct {
  <type> field1;
  <type> field2;
  <type> field3;
} <compositeType>;

//usage
<compositeType> var1;
```

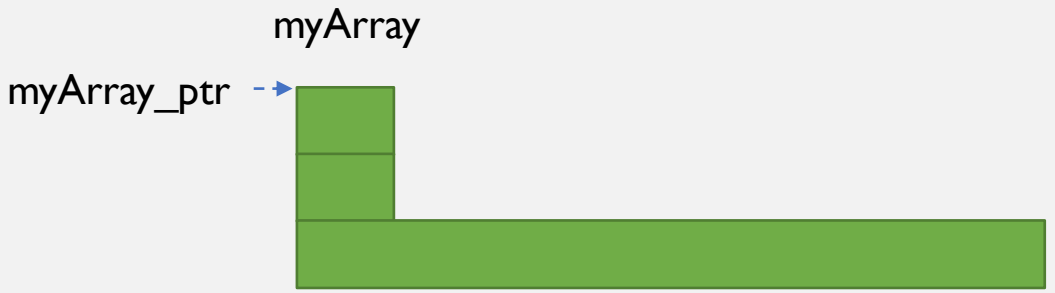
```
typedef struct {
    unsigned int capacity;
    unsigned int used;
    double elements[10];
} doubleArrayWithSize;
```

```
doubleArrayWithSize myArray;
```

Composite Data Type – Access

- To access the fields (variables inside the composite data type), use the **.** (dot) operator
  - To access the fields when the composite variable is a pointer, use **(*...).** or **->**

```c
//initialization
doubleArrayWithSize myArray = {
    .capacity = 10,
    .used = 0,
    .elements = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

//access
myArray.elements[0] = 1.2;
myArray.used = 1;
printf("Value of first element: %f\n", myArray.elements[0]);

doubleArrayWithSize* myArray_ptr = &myArray;
(*myArray_ptr).elements[1] = 2.4;
myArray_ptr->used = 2;
printf("Value of second element: %f\n", myArray_ptr->elements[1]);
```




```
Value of first element: 1.200000
Value of second element: 2.400000
```

Functions with Structs

- Just like any other data types, structs can also be passed to a function as a parameter

```c
void insert(doubleArrayWithSize* da, double value) {
    //insert value if there is space
    if (da->used < da->capacity) {
        da->elements[da->used] = value;
        da->used++;
    }
}
```

pass-by-reference to have access to the actual struct variable

```c
int main() {
    doubleArrayWithSize myArray = {
        .capacity = 10,
        .used = 0,
        .elements = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    };

    printf("Capacity: %d\nAvailable: %d\n", myArray.capacity, myArray.capacity-myArray.used);
    insert(&myArray, 1.2);
    printf("Capacity: %d\nAvailable: %d\n", myArray.capacity, myArray.capacity-myArray.used);

    return 0;
}
```

```
Capacity: 10
Available: 10
Capacity: 10
Available: 9
```

- Later we'll learn how to return multiple values inside a struct from a function (recall a function only returns 1 value)

Enum (Enumeration)

- Sometimes we want to have user defined types to store a few values that make sense together, for example:
  - 4 suits in playing cards (with names spades, hearts, clubs, diamonds)
  - 7 days in a week (with names Monday, Tuesday, …etc.)
  - 12 months in a year (with names January, February, March, …etc.)
  - Game state (with names started, paused, gameover, …etc.)
- Enums allow us to define a type and assign **names** to integers
  - So we can use those more meaningful names in the code
  - If no specific value is set for the first name, C assigns 0 to it, and adds 1 to the next, unless it is set explicitly

Enum – Syntax

- Basic syntax is to let C start assigning from 0 and add 1 for the next

```
enum <EnumType> {
  name1,
  name2,
  name3
};

//usage
enum <EnumType> var1;
```

```
enum Suits {Spades, Hearts, Clubs, Diamonds};
```

```
enum Suits mySuit;
```

- A more common way is to "typedef" it so we just need to use the type name itself

```
typedef enum {
  field1,
  field2,
  field3
} <EnumType>;

//usage
<compositeType> var1;
```

```
typedef enum {Spades, Hearts, Clubs, Diamonds} Suits;
```

```
Suits mySuit;
```

Enum – Usage

- To use an enum, simply write its named value (this means the same value cannot be used in another enum type)

```
//initialization
Suits mySuit = Spades;
printf("mySuit is %d.\n", mySuit);

if (mySuit == Spades) {
    printf("mySuit is Spades.\n");
} else if (mySuit == Hearts) {
    printf("mySuit is Hearts.\n");
} else if (mySuit == Clubs) {
    printf("mySuit is Clubs.\n");
} else {
    printf("mySuit is Diamonds.\n");
}
```

Can you replace this with switch?

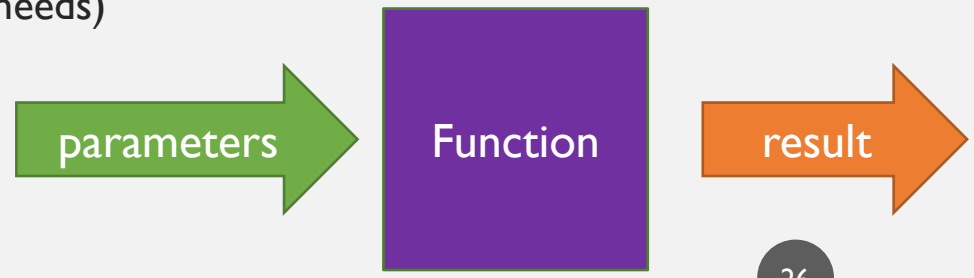
```
mySuit is 0.
mySuit is Spades.
```

Internally, Spades has the value 0, Hearts has the value 1, Clubs has the value 2, Diamonds has the value 3

So
Suits yourSuit = Spades + Hearts;
works

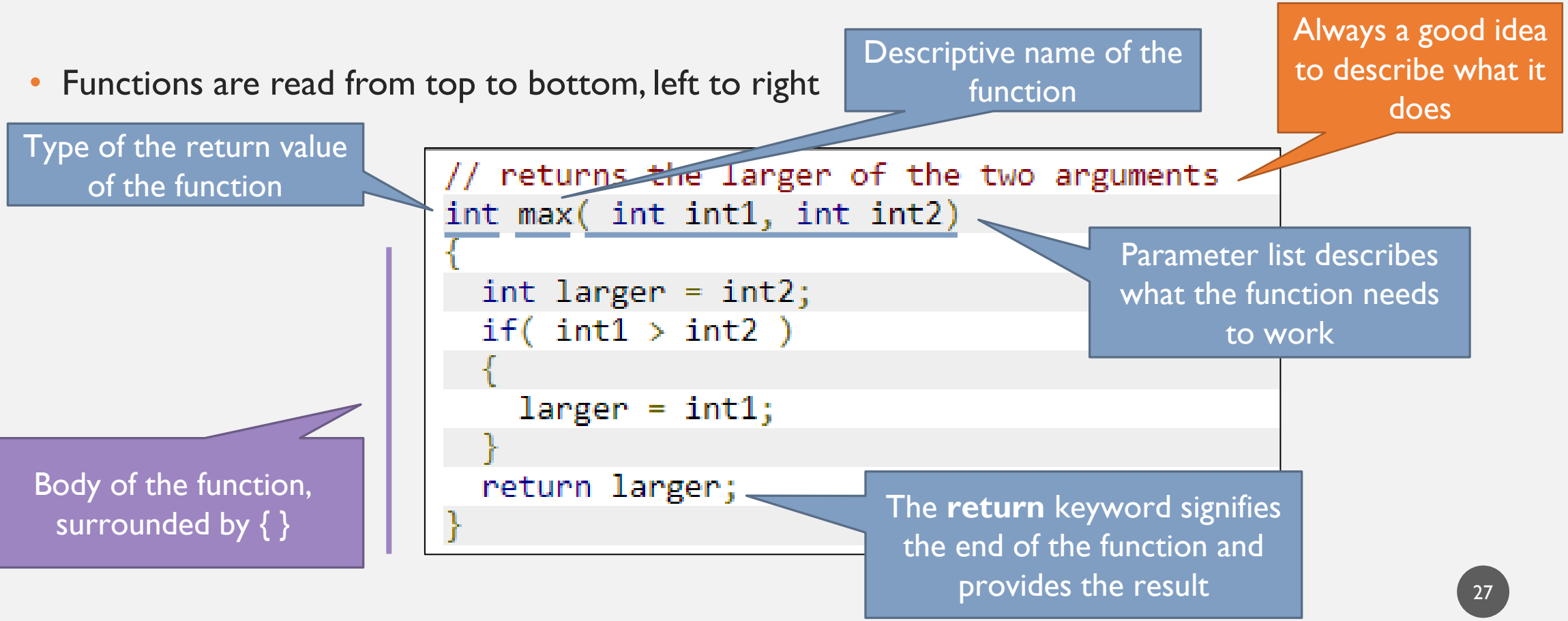
Functions

- A function is a self-contain block of code that can be reused as components of larger programs
  - Functions can call other functions, including themselves (if so we call them recursions or recursive functions)
  - Functions can be written separately (in different files)
- A function works like a **black box**
  - The caller doesn't need to know what is done inside the function, all it needs to know is:
    - Name of the function (so it can call the function)
    - Parameters needed by the function (so it can give what the function needs)
    - Return value of the function (so it can store and use the value)



Function Definition

- Functions are read from top to bottom, left to right



The return Keyword

- Functions are conceptualized as something that perform a task and "return" you with the result
  - Hence, the word **return** is used to signify the function has finished its task and has the resulting value ready
  - It therefore also has the "side-effect" of terminating the function and "returns" to the code execution where it is called
- A function can have multiple return statements for different results
  - each from a different way of calculation (case) in the function
  - each representing a different status of the function, e.g., successful, failed due to reason 1, failed due to reason 2, …etc.
- A special type of function: void function, has no returned values, but can still use the **return** keyword to terminate

Calling Functions

- To start, functions are defined in the same file as the main
- Whenever you want to call (use) a function, write its name & provide what it needs to work
- When the program sees the call, it will look up the function definition, run the code there, & replace this call with the returned value

```
int main( void )
{
  int num1 = 11;
  int num2 = 12;
  printf( "The max of our numbers is %d.\n", max( num1, num2 ));
  printf( "The min of our numbers is %d.\n", min( num1, num2 ));
  return 0;
}
```

Name of the function

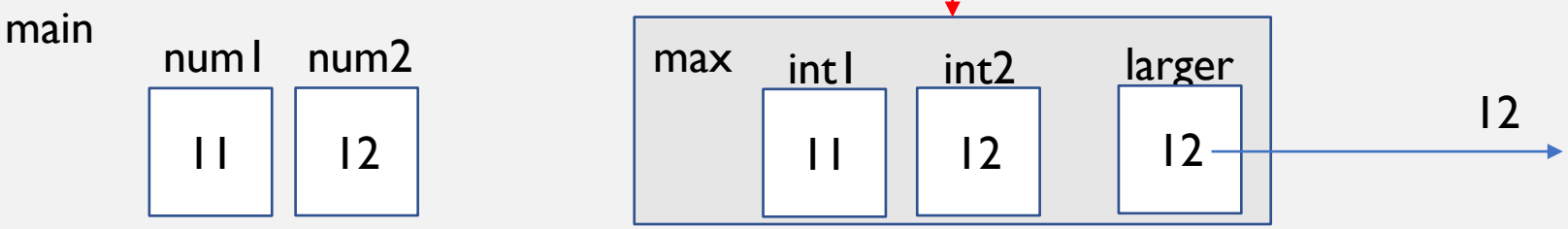
This function needs 2 parameters to work

Tracing Functions

- A useful way to understand how functions work is to trace them by hand

```c
// returns the larger of the two arguments
int max( int int1, int int2)
{
  int larger = int2;
  if( int1 > int2 )
  {
    larger = int1;
  }
  return larger;
}
```

```c
int main( void )
{
  int num1 = 11;
  int num2 = 12;
  printf( "The max of our numbers is %d.\n", max( num1, num2 ));
  printf( "The min of our numbers is %d.\n", min( num1, num2 ));
  return 0;
}
```



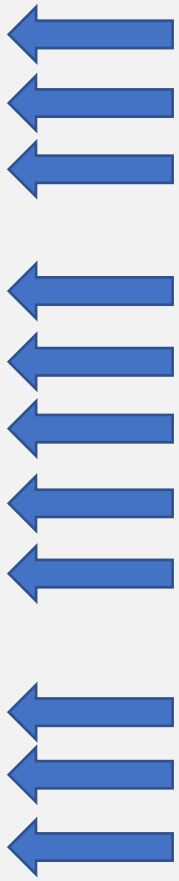
Execution (Call) Stack

- The way we trace the function is a simplified version of how functions actually get executed (called)
  - The program uses a data structure to store the information about the functions to keep track of things
  - It is called a stack because everytime a function is called, information about this function is "stacked"

- Each piece of information (frame) contains the following:
  - parameters of the function: values passed to the function by the calling function
  - local variables: variables declared inside the function definition
  - return value: value that can be accessed by the calling function when the function completes (returns)
  - return address: when the function completes (returns), which calling function should the control go back to

Execution Stack Example

```
int bar (int size) {
    int i = size+1;
    return i;
}
int foo (int n) {
    int ret = 3;
    bar(4);
    bar(8);
    return ret;
}
int main() {
    foo(5);
    return 0;
}
```



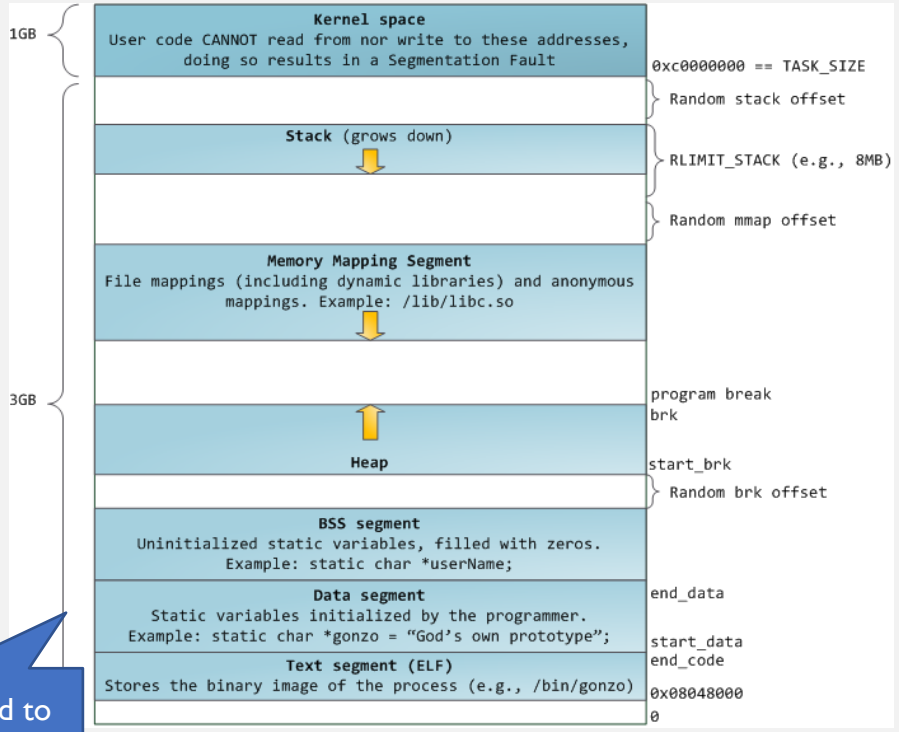
bar()
params: int size = 8;
variables: int i= 9;
return value: ?
return address: 0x9965ff

foo()
params: int n = 5;
variables: ret = 3;
return value: ?
return address: 0x9378ad

main()
params: --
variables:…
return value: ?
return address: 0x128fbad

Execution Stack And Scope of Variables

- All local variables of a function are stored in the corresponding stack-frame that gets created when called
  - when the function completes, the stack-frame is freed up and these variables become unavailable (to be used by another stack-frame)
- Dynamically allocated memory obtained from malloc() are stored in a separate memory space and do not get freed up automatically along with the stack-frame of the completed function
  - Must free update this memory ourselves when we don't need them anymore
- Local variables are stored on the stack
- Dynamically allocated memory are store on the heap



FYI only, no need to memorize ☺
Also varies OS to OS

Source: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Today's Review

- Conditionals
  - if-statements
    - One more variable type: bool
    - 3 kinds (forms): if, if-else, if-else if-else (multiple else if's)
    - "Closest-if Rule"
  - switch-statements
    - use breaks to prevent "fall-through"
- Composite data types (struct & enum)
  - user-defined ways to represent entities (struct) & small set of meaningful values (enum)
- Functions

Homework!

- Continue to read the sections in Ch. 2 & Ch. 5 of the Effective C book
  - Pay attention to the section on today's topics: enum Types, structures, Selection statements
- For the switch example in p14 (determining grades), write the equivalent using if-else if-else statements
- For typedef enum {North=0, East=90, South=180, West=270} Direction;
  - What value does C assign to the name South? How do you print this value out?
- Design and Define a struct that has 1) a struct, and 2) an enum as its fields, besides other built-in types
- Learn about the switch statement (bottom of page): http://www.cplusplus.com/doc/tutorial/control/
- Stay tuned to Assignment 1. Will be released today or tomorrow on Canvas!