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

Fact of the day

TYPEWRITER is the **longest word** that you can write using the letters **only on one row of the keyboard** of your computer

Recap from Last Lecture

- Conditionals (if-statements, switch-statements)
  - Also known as branching, allows programs to make decisions based on states (conditions)
  - One more variable type: bool – only has values true/false, can be used as a “flag”
- Composite data types
  - struct – putting multiple variables together, enum – defining possible values
- Functions
  - Self-contained bloc of code that can be reused as components of larger programs

Review from Last Lecture (1)

- Switch statements are another control structure that decide which block of code to execute based on comparison
  - It works like an if-else if-else structure, except each case (condition) must be an equality check of values (e.g., int/char)

Get an integer value for equality check

Decide what to do based on the value

```
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");
}
```

=

```
if (marks >= 100 || marks >= 90) {
    puts("YOUR GRADE : A");
} else if (marks >= 80) {
    puts("YOUR GRADE : B");
} else if (marks >= 70) {
    puts("YOUR GRADE : C");
} else {
    puts("YOUR GRADE : Failed");
}
```

Can also just use marks >=90

Review from Last Lecture (2)

- 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?
    - 180 (you can try using the printf function with %d to print the value of South to verify)
- Design and Define a struct that has 1) a struct, and 2) an enum as its fields, besides other built-in types
  - One example can be a struct representing a chess set: 1)pieces, each is a struct representing its colour, type, and position, and 2) whose turn is it

```
typedef enum {Black, White} Colour;
typedef enum {Rook, Knight, Bishop, Queen, King, Pawn} Type;
typedef struct {
    Colour colour;
    Type type;
    char position[2];
} Piece;
```

```
typedef struct {
    Colour currentTurn;
    Piece pieces[32];
} ChessSet;
```

Today

- Type casting
- Functions in other files
- Preprocessor directives (Macros)
- Global/Local/Static variables
- Memory allocation

Type Casting

- C allows conversions between different types of variable so that a variable can act like another in one operation
  - It is not changing the original value, just make it behave like another type during the cast
  - Examples of type casting: int to float, int to long, short to int

```
int anInt = 0;
float aFloat = (float) anInt;
long aLong = (long) anInt;

short aShort = 0;
int anotherInt = (int) aShort;
```

  - If a type is casted to another type with less storage space or with less precision, information will be lost
    - For example, float to int, int to char, long to short

Notes about Type Casting

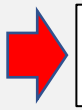
- Strongly-typed programming languages has a type for each variable, enforcing what operations are possible

```
int anIntArray[10];
anInt = anIntArray;
```

```
main.c:20:11: warning: assignment to 'int' from 'int *' makes integer from pointer without a cast [-Wint-conversion]
   20 |     anInt = anI:The average between 3 and 4 is 3
      |           ^
```

- Sometimes the compiler will do it for you automatically (implicit conversion) but depending on the direction some information might be lost → make sure you know what you are doing!

```
long aLong = 9876543210;
int anotherInt = aLong;
```



```
anotherInt has the value 1286608618
```

- A typical use of type casting is to force the conversion to happen

```
int firstNumber = 3, secondNumber = 4;
float average = (float) (firstNumber + secondNumber) / 2;

printf("The average between 3 and 4 is %f\n", average);
```

```
The average between 3 and 4 is 3.500000
```

Where to Place your Function Definitions?

- A function needs to be "known" before it can be called

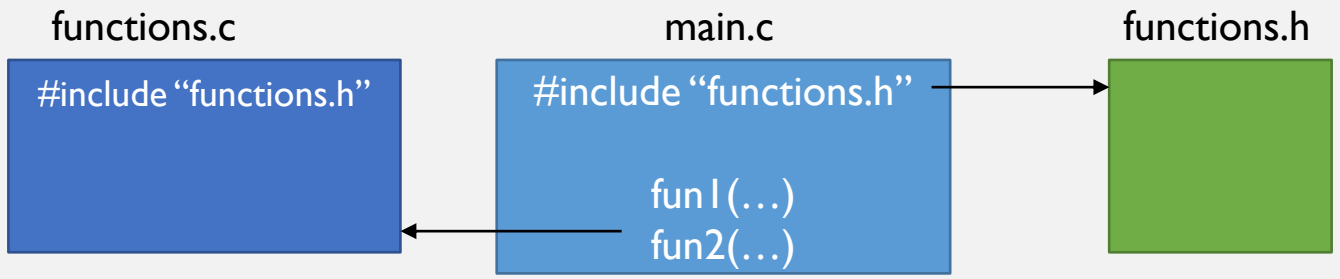
```
#include<stdio.h>

int main() {
    doSomething();
    //rest of code
```

```
lib/gcc/x86_64-alpine-linux-musl/9.3.0/../../../../x86_64-alpine-linux-musl/bin/ld: /tmp/cclNFgMe.o: in function `main':
main.c:(.text+0xe): undefined reference to `doSomething'
collect2: error: ld returned 1 exit status
```

- One way is to put the function definitions before main
- Another way is to put the function header (similar to a declaration) before the code that calls the function, and the function's implementation later in the file or in another file
  - Allows better code management (e.g., easy to swap out the implementation, group related functions)
  - Allows work distribution (e.g., Programmer A works on all input functions, Programmer B on output)

How Does Functions in Other Files Work?

- A further step is to put all the function headers in a header file, and other files include it
  - Keeps user-written file short (just need 1 line of #include for many related functions)
  - This is what we have been doing with stdio.h, string.h, and stdbool.h
  - We will also be doing this in our assignments



Compilation of C

- Invoking the compiler **gcc** on the source code file causes a few things to happen:
  - the C preprocessor runs by scanning the source code file and make some changes to the code, including removing things like comments and newlines, and expanding macros by replacing it with something else or carrying out some actions
  - the C compiler then runs by translating the C code into assembler code specific to your system architecture
  - the C compiler then further translates the assembler code into binary format that can be read by the computer directly
  - the C compiler then links other object files and libraries to create an executable program

Preprocessor Directives (Macros)

- These are "orders" to the preprocessor during compilation
  - each directive must be specified in a single line, starts with a #, and does not end with a ; (they are not statements)
- Some useful directives are:
  - #include – replaces this line with the whole content of the specified file, typically a header file
    - #include "some_file" makes the compiler look for some_file in the same directory where the directive is at
    - #include <some_file> makes the compiler look for some_file in the directory where it is configured to look for standard header files (e.g., stdio.h, string.h, stdbool.h)

Preprocessor Directives (Macros) (Cont'd)

- #define – defines a macro so that whenever it is seen in the code it gets replaced with its corresponding value
  - Typically used to define numerical constants, for example, #define MAX_NAME_LENGTH 256 allows this code to run

```
char student1Name[MAX_NAME_LENGTH];
printf("Maximum number of characters in a name: %d\n", MAX_NAME_LENGTH-1);
```

  - Can also be used as textual substitutions

```
#define MY_FRAC 70/14
#define SQR(a) a*a

int main() {
    float x = MY_FRAC; // replaced by float x = 70/14;
    int y = SQR(5); // replaced by int y = 5*5;
    int z = SQR((5+2)); // replaced by int z = (5+2)*(5+2);
    int w = SQR(5+2); // replaced by int w = 5+2*5+2;
```

More Preprocessor Directives

- The **#ifndef – #define – #endif** directive trio means the following:
  - if something is not defined, include the code between #ifnded and #endif for compilation, otherwise skip the code
- This is used in header files so that multiple includes of the same header file won't cause the compiler to include the same code multiple times (e.g., both the test and assignment files for A1 include the stdio library)
  - This use of directives is called "include guards"
  - By convention, the token used in the #define directive as include guard is _FILE_NAME_H_

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_

/* code */

#endif
```

Global VS Local

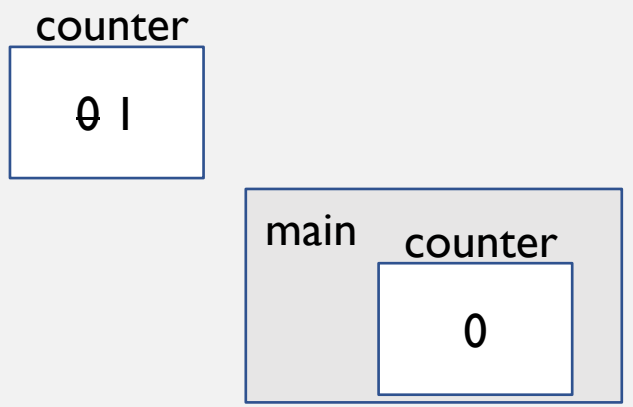
- When defining **functions** and **composite data types**, we put them outside of main, so they can be used anywhere
  - We call these definitions "global"
- We can also declare variables outside of main so they can be used anywhere – global variables
  - Generally this is not a good idea because any code can change their values, and to understand code that uses global variables mean you have to find all the global variables first
  - If you really want to use global variables, declare them as constants, and give them meaningful names (e.g., MAX_INT_SIZE)

- Rule-of-thumb: the scope of a variable (when it can be used) is within the { } where it is declared/initialized

Global VS Local

- If there is a name conflict (same variable name used in a local scope), the local variable is used

```c
#include <stdio.h>

int counter = 0; // initialize the global variable to zero

int main() {

    printf("global counter %d\n", counter); // prints 0
    counter++;
    printf("global counter %d\n", counter); // prints 1
    int counter = 0; // local variable
    printf("local counter %d\n", counter); // prints 0

    return 0;
}
```




```
global counter 0
global counter 1
local counter 0
```

Static Variables

- There are times where a variable that preserves its value in its previous scope is useful, for example, number of times a function is called while the program is running
- Declaring a variable as static within a function (aks static local variable) allows this to happen

```c
#include <stdio.h>

void test_static_count() {
    static int count = 0; // initialized only once, and must be a constant
    count++;
    printf("count = %d\n", count);
}

int main() {
    test_static_count(); //prints 1
    test_static_count(); //prints 2
    test_static_count(); //prints 3

    return 0;
}
```

```
count = 1
count = 2
count = 3
```

Memory Allocation

- Let's consider writing a program that asks the user for:
  - a number (e.g., N) indicating how many data input the program needs to process
  - N data inputs (e.g., N integers)
- Issue: we don't know how many inputs in advance, we cannot declare an array of unknown length – and we don't want to create unnecessary variables
- Solution: create variables when needed during runtime
- C has a mechanism for doing so: Dynamic Memory Allocation
- The library containing the functions doing that is **stdlib.h**

The malloc function

```
void* malloc (size_t size);
```

- Takes in 1 parameter indicating the requested size in bytes (typically returned by the **sizeof** function)
- Returns the memory address of the space reserved for the size requested (then stored by a pointer variable)
  - If such space cannot be reserved, the function will return the NULL pointer (that's how we check if the call is successful)

Need to include the **stdlib.h** library

```
int* int_ptr = malloc(sizeof(int));
int* intArr_ptr = malloc(sizeof(int) * 10); //array of 10 ints

if (intArr_ptr == NULL) {
    printf("Cannot reserve memory for array of 10 ints.\n");
}
```

Optional: cast the returned value of malloc to the expected point type, e.g., int* p = (int*) malloc(…)

The realloc function

```
void *realloc( void *ptr, size_t new_size );
```

- realloc returns the same memory address from the heap with the specified size, but if it can't, it will:
  - allocate space somewhere else, copy the content over, free the original space, & return the address of the new space (if no new space is available, it'll return NULL)
  - Usage: realloc(ptr, sizeof(int) * newLength)

- Compare realloc with malloc which returns a newly found memory address from the heap with the specified size, or NULL if fails
  - Usage: malloc(sizeof(int) * length)

The free() Function

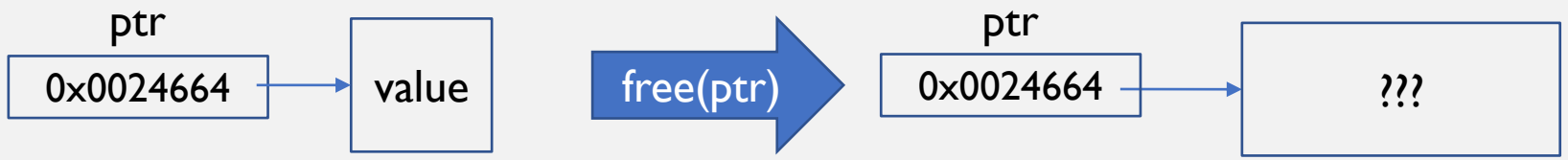
- When memory is allocated using malloc, it has to be manually released (there is no scoping)
  - If we keep requesting memory but not releasing them, program will eventually run out of available memory, and in some cases, causes memory leak (memory reserved but not accessible by anything)
- Use the free() function to release memory when it is no longer needed
  - Should also set the pointer variable to the NULL pointer because it will be storing an address that is no longer valid

```
free(int_ptr);
int_ptr = NULL;
free(intArr_ptr);
intArr_ptr = NULL;
```

- Rule-of-thumb: when there is a malloc, there is a corresponding free

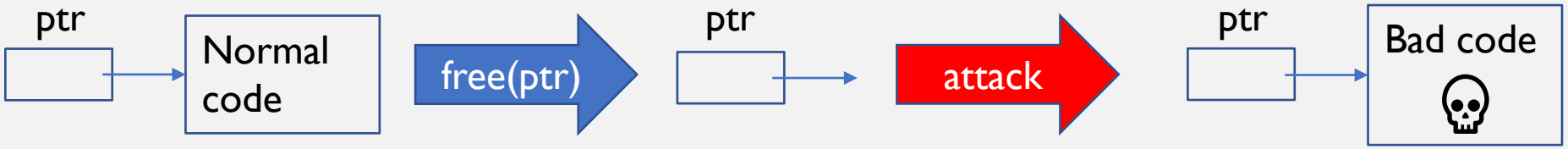
The Dangling Pointer Issue

- When free() is called, the memory pointed to by the parameter is de-allocated, but the pointer variable **will still be storing the address**, this situation is called having a **dangling pointer**
- Typically happens when
  - free() is called but the pointer is not set to NULL afterwards
  - Multiple pointers are pointing to the same memory, free() is called with one of them
- This is an issue because if later this pointer is used, the dereferenced value can be anything (unused, or allocated to some other dynamic variable requests)



Side Note: Use-After-Free Vulnerability

- Use-After-Free (UAF) is a vulnerability of a software related to incorrect use of dynamic memory during program operation. If after freeing a memory location, a program does not clear the pointer to that memory, an attacker can use the error to hack the program. (source: Kaspersky)
- Basically, this kind of attack substitutes malicious code to a heap memory that is previously freed but the pointer to this address has not been cleared (aka dangling pointer)
- Even modern programs can suffer from this (e.g., recent Chrome zero-day exploits, iOS exploits)



- Rule-of-thumb: always set the pointer to NULL when you free the memory it is pointing to

Dynamic Memory Allocation And Arrays

- Since arrays are just a contiguous block of memories for a collection of variables, and array variables in C are just pointers to the first element, you can create arrays of any size

```c
int arraySize = 0;
printf("How many numbers do you have? ");
scanf("%d", &arraySize);
int* arrayPtr = malloc(sizeof(int) * arraySize);
printf("Enter one number at a time, press enter after each entry: ");
for (int i=0; i<arraySize; i++) {
    scanf("%d", &arrayPtr[i]);
}
printf("You have entered: ");
for (int i=0; i<arraySize; i++) {
    printf("%d ", arrayPtr[i]);
}
```

```c
//no more use of the array
free(arrayPtr);
```

Creating Arrays in Functions

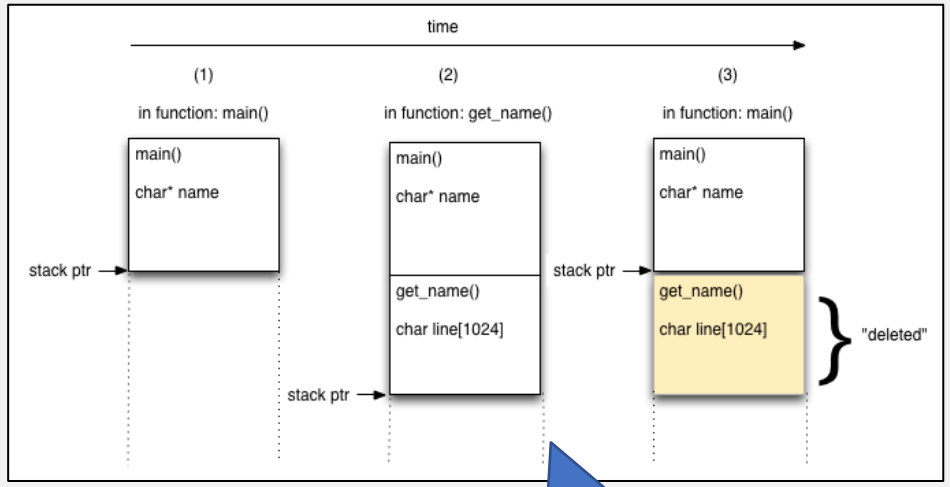
- Since functions can return values, including addresses, and array variables in C are just pointers to the first element → we can use functions to create arrays of any size and return them
- For example:
  Write a function that gets a parameter int n and returns the array of ints of length n, where array[i] = i^2

This is just to show you arr1 can exist but must not be returned… you can remove it

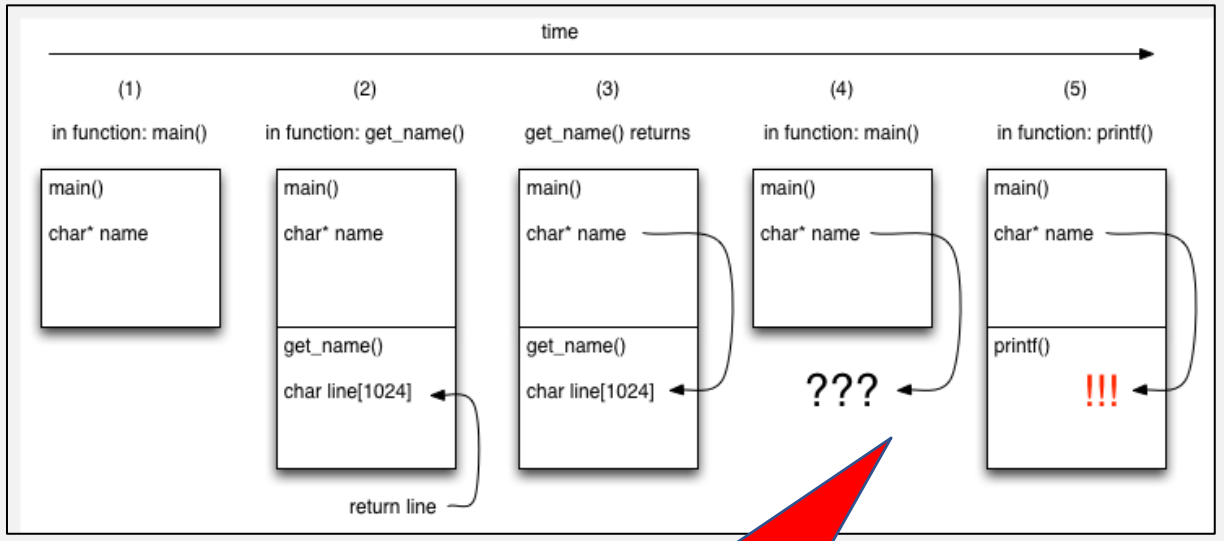
```
int* generateSquares(int n) {
    int arr1[15]; //static arrays created in function must not be returned
    int* arr2 = (int*) malloc(sizeof(int)*n);
    for (int i=0; i<n; i++) {
        arr2[i] = i*i;
    }
    return arr2; //this is an address
}
```

Why Must We Not Return Local Arrays?

- Recall each time a function is called a stack-frame is added containing local variables and is removed when done



Stack-frames come & go



Attempt access to unmanaged memory

Unsafe Variable-Length Arrays (VLA)

- Some compilers allow declaring arrays of variable lengths

```
//don't do this
int unsafeFunction(int n) {
    int arr[n]; //size unknown at compile time
}
```

- Don't do this because it is inherently unsafe… each stack-frame needs to know all its local variable in advance so proper space can be allocated → this also leads to more optimized programs
  - it might happen that the function is called with a very large n and memory is not properly allocated due to unavailability
- Rule-of-thumb: If we don't know the length of array when writing the code, use dynamically allocated memory

Today's Review

- Type casting
  - Convert between types to allow type-specific behaviour, might lose precision
- Functions
  - Reusable code blocks, can be defined in a different file as long as the header is available
- Preprocessor directives (Macros)
  - Instructions for the C compiler to setup the code and collect resources
- Global/Local/Static variables
  - Availability of variables, conflict resolution
- Memory allocation
  - Allows programs to request (and manage) memory during runtime, needs to be careful

Homework!

- Read Ch. 6 of the Effect C book
- Investigate how to use request memory for structs
- In p17, what happens if the keyword static is removed from the test_state_count function?
- Look up other memory allocation functions: calloc, memcpy, memset
  https://www.cplusplus.com/reference/cstdlib/
  https://www.cplusplus.com/reference/cstring/ (strangely, memcpy & memset are from the C string library!)