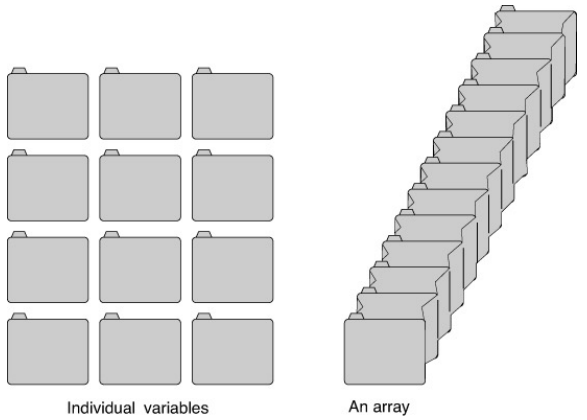
COMPSCI2030 Systems Programming

Program Structure

Yehia Elkhatib

University of Glasgow

Arrays



- Multiple related elements of the *same* type
- Accessed using a single name and an index (aka subscript)
  - starts at `0`

```
_Bool Spartans[300];
Spartans[0] = 1;
Spartans[1] = 0;
Spartans[2] = 1;
```

- The elements in an array are stored next to each other in memory
- Arrays that are stored on the stack must have a fixed size
  - so that the memory is automatically managed
- We can still use dynamic arrays, i.e. their size can change
  - not managed automatically

Array declaration

- Like any variable, arrays must be declared
- Declaration must include data type and size (number of elements)

```
int data[2];
data[0] = 3;
data[1] = 7;
```

- Size could be inferred from initialisation

```
int data[] = {3, 7};
```

- Multidimensional arrays
  - as many dimensions as can be held in your memory
  - stored in a row major format, i.e. rows are stored after another

```
int array[4][3] = { { 1, 2, 3 } , { 4, 5, 6 } ,{ 7, 8, 9 } , { 10, 11, 12 } };
int array[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }; //same as above
```

Strings

- A sequence of characters, i.e. an array of `char`'s
  - also a `char*` (more on this later in the course)
- Quotation marks

```
char greeting[] = "Hello World";
char greeting[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\0'};
```

  - we use doubles " for string literals and singles ' for `char` literals
- Strings are terminated by a special character `'\0'`
  - this is added automatically for string literals
  - if we forget it, you would read from / write to memory all contents until you hit the next bit pattern equivalent of `'\0'`!
- To print a string with `printf` we use the `%s` formatting character

```
printf("%s\n", greeting);
```

String Manipulation

- A lot in string.h
  - `strlen` – get length
  - `strcpy` & `strncpy` – copy contents
  - `strcat` & `strncat` – concatenate
  - `strcmp` & `strncmp` – compare two strings, character by character
  - `strchr`, `strrchr`, `strcspn`, `strspn`, `strpbrk`, `strstr` – search string

- Please look them up on WikiBooks

Structs

- Another data structure that combines multiple elements
- It consists of a set of members of (potentially) different types
- Members are accessed using the . notation
  - like public class members in Java
- Members are stored adjacently in memory in the order of definition
- The type of a struct is written `struct name`
- ...but we can use `typedef` to shorten it

```c
struct point {
  int x;
  int y;
};
int main() {
  struct point p = {1, 2};
  printf("x = %d\ny = %d\n", p.x, p.y);
}
```

```c
typedef struct { int x; int y; } point;
int main() { point p = {1, 2}; /* ... */ }
```

Functions

```c
int max(int lhs, int rhs) {
    if (lhs > rhs) {
        return lhs;
    } else {
        return rhs;
    }
}
```

- A self-contained section of the code to carry out a task
  - modular and reusable
- A very important abstraction mechanism in programming
- A function definition in C looks like this:

```
return_type function_name(param_type param_name, …) { body }
```

  - *return type* - the data type of the value that will be returned after evaluating the function
    - if no value is to be returned, the special type `void` is used
  - *function name* - a reference, ideally descriptive of the behaviour of the function
  - *parameter list* - specifies the data type and name of each parameter the function expects
  - *function body* - a block containing the code to be executed when calling the function
- To call a function: we provide an argument for each parameter and capture / process the return value

Function declaration and definition

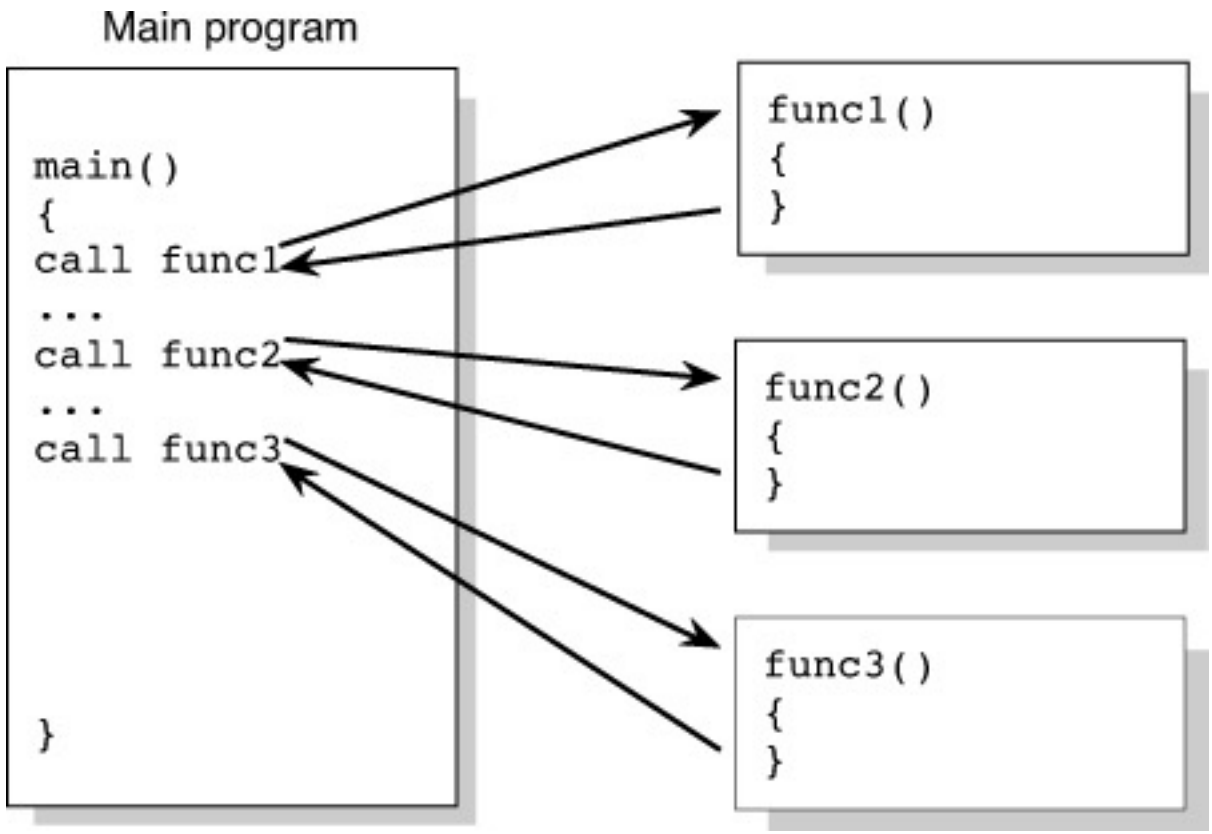
- A *function declaration* only specifies how it can be used

```
int max(int lhs, int rhs);
```

- Function declarations are important for writing modular software
  - it allows to separate the interface (the declaration) from the implementation (the definition)

How function calls work

- Statements in a function are not executed until the function is called by another part of the program
- When a function is called, the statements in the function then execute (using the arguments)
- When the function's statements finish, execution passes back to the same location in the program that called the function



Call by value

```c
#include <stdio.h>

void set_to_zero(int x) {
    x = 0;
}

int main() {
    int y = 42;
    set_to_zero(y);
    printf("%d\n", y);
}
```

- How are arguments passed to functions when we call them?
  - The value 42 will be printed, because when we call a function we pass *a copy of the arguments* to it
  - x and y are stored at different memory locations
  - When `set_to_zero` is called the value of y is copied into the memory location of x
- We call this *call-by-value* as the argument value is evaluated and stored in the parameter variable
- All function calls in C pass arguments by value…

Call by reference

- … except for arrays
- Copying an array could be very expensive (memory-wise)
- Instead, the *address* of the first element is passed by value
- Hence, any changes inside the function happen to the original data
  - i.e. changes to array elements are now visible *outside* the function
- This special treatment of arrays will make more sense when we learn about pointers

Lexical Scoping

- Each pair of curly braces { } is a block and introduces a *lexical scope*
- Variable names must be unique in the same lexical scope
- For multiple variables with the same name, the variable declared in the innermost scope is used
- Which values will be printed?

```c
#include <stdio.h>

int main() {
  int i = 5;
  {
    int j = foo(i);
    printf("%d\n", j);
  }
}
```

Then 6

```c
int foo(int i) {
    int j = i;
    {
        int i = 0;
        int j = i + 2;
        printf("%d\n", j);
    }
    return j + 1;
}
```

First 2

Variable Lifetime

- Variables are stored at locations in memory that do not change over their *lifetimes*
  - This depends on *how* the memory for the variable was allocated. There are 3 cases:

Stack

1. *automatic*:
   - Declared locally in a block (i.e. inside a pair of {})
   - Their lifetime ends at the end of the block
   - All variables we have seen so far fall into this category

```
int main() {
  int x = 42;
} // end of the block = end of the lifetime of x
```

2. *static*:
   - Declared with the `static` keyword
   - … or defined at file-level outside all blocks
   - The lifetime is the entire execution of the program

```
int foo() {
  static int count_calls_to_foo = 0;
  count_calls_to_foo++;
  return count_calls_to_foo;
} // variable continues to live
```

Heap

3. *allocated*:
   - Explicit request of using dynamic memory allocation functions, such as `malloc`
   - We manage the lifetime of these variables ourselves (next lecture!)

make

- GNU make is the most popular build system for C programs
  - automates the process of compiling programs
  - alternative build systems: Maven, Bazel, Ninja, …
- A Makefile has the following structure:

```
program : source.c
    clang -Wall -Werror source.c -o program
    # ^ this space must be a single tab character!
```

make

```
program : source.c
    clang -Wall -Werror source.c -o program
    # ^ this space must be a single tab character!
```

For more details
`man make`

- The first two lines form a rule that explain how a target is built
- The first line is the dependency, made up of "[targets] : [sources]"
- The target(s) depend on the source(s)
  - elements in each list are separated by a space
- The following line(s) are the action
  - i.e. commands defining how the target is built
  - need to start with 1 tab
- Makefiles are white-space sensitive!
- Running `make` will execute the first rule in the current directory
- `make target` will execute the rule to make the named target

Question time

1. What happens if I use an index on an array that is larger than the number of elements in the array?
2. An array is declared using `int array[2][3][4][5];`
   a. How many total elements does the array have?
   b. What would be the name of the tenth element in the array?
3. What must be the first line of a function definition, and what information does it contain?