CS 220: Introduction to Parallel Computing

Arrays

Lecture 4

Note: Windows

- I updated the VM image on the website
- It now includes:
  - Sublime text
  - Gitkraken (a nice git GUI)
  - And the git command line tools

Today's Agenda

- Argument Passing in C
- Compilation Phases
- Arrays

Today's Agenda

- **Argument Passing in C**
- Compilation Phases
- Arrays

Argument Passing Conventions

- Coming from the Java or Python world, we're used to passing **inputs** to our functions
- The result of the function is given to us in the **return value**
- There are situations where this convention does not hold, but it's less common
- This is **not** the case with C…

An Example

```c
/* Here's a function that increments
 * an integer. */

void add_one(int *i)
{
   *i = *i + 1;
}

int a = 6;
add_one(&a); /* a is now 7 */
```

C "In/Out" Arguments

- In C, some of the function arguments serve as **outputs**
  - Or in the example we just saw, the function argument is **both** an input and an output!
- Some API designers even label these arguments as "in" or "out" args (example from the Windows API):

```
BOOL WINAPI FindNextFile(
 _In_ HANDLE hFindFile,
 _Out_ LPWIN32_FIND_DATA lpFindFileData
);
```

- Why?

Error Reporting

- One reason for this is C does not have exceptions
- Problem in a Java/Python function?
  Throw an exception!
- In C, the return value of functions often indicates success or failure (bool)
  - Or a maybe something in between (int) – status code
- Functions don't **have** to be designed this way, but it's a very common convention

Efficiency

- Return values have to be copied back to the calling function
  - Say my function returns a bitmap image. The entire thing is going to get copied!
- In a language that focuses on speed and efficiency, updating the values directly in memory is faster
- Imagine transferring lots of large strings, objects, etc. around your program, copying them between functions each and every time

C Argument Passing

- So remember: the return value in C **might not** actually be the result of the function
- It may just be an error code or status code
- It might just be a Boolean
  - True means success, false means failure
- Or maybe one of the function inputs was modified directly instead
  - Efficiency benefits

Void Argument (1/3)

- In C, there's a difference between `function()` and `function(void)`
- Void arg: the function takes **no arguments**
- Empty arg list: the function **may or may not** take arguments
  - If it does, they can be of any type and there can be any number of them

Void Argument (2/3)

- Why is this important?
- First, to understand older code
  - From the C11 standard:
    - "The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature."
- Second, this may lead to incorrect function prototypes or passing incorrect args in your code

Void Argument (3/3)

So, to sum up:

```c
/* Takes an unspecified number of args: */
void function();

/* Takes no args: */
void function(void);
```

Today's Agenda

- Argument Passing in C
- **Compilation Phases**
- Arrays

Recall: Phases of C Compilation

1. ***Preprocessing***: perform text substitution, include files, and define macros. The first pass of compilation.
   - Directives begin with a #
2. ***Translation***: preprocessed code is converted to machine language (also known as **object code**)
3. ***Linking***: your code likely uses external routines (for example, printf from stdio.h). In this phase, libraries are added to your code

Stepping Through Compilation

- When we compile our source code, we get an output binary that is ready to run
  - The steps are mostly invisible to us

- We can ask the compiler to only execute a subset of its compilation phases
  - Let's do just that!

Preprocessing

- We can ask gcc to only perform the preprocessing step using the -E flag:
  - gcc -E my_program.c
- This will print the preprocessed file to the terminal
- We can write this output to a file by redirecting the stdout (**standard output**) stream:
  - gcc -E my_program.c > my_program.pre
- … And view with a text editor

Translating to Assembly Code

- We can also view the **assembly** code generated by the compiler
- gcc -S my_program.c
  - Produces my_program.s
- This representation is very close to the underlying **machine code**
- For a reference on x86-64 processor assembly:
  - https://web.stanford.edu/class/cs107/guide_x86-64.html

Producing Object Code

- Finally, we can produce the **machine code / object code** representation of the program
- gcc -c my_program.c
  - Produces my_program.o
- We can view this with a **hex editor**
  - hexdump -C my_program.o

Linking

- Finally, our object code is linked against the other necessary libraries to create an executable
- Nothing to inspect here, but we can always view the output binary in a hex editor:
  - hexdump -C my_program

Today's Agenda

- Argument Passing in C
- Compilation Phases
- **Arrays**

Arrays

- In C, arrays let us store a collection of values of the **same** type
- They are similar to the arrays in Java, and roughly analogous to the lists in Python
    - However, Python lets us store values of **different** types:
        - my_list = [1, 6.8, “San Francisco”]
- In C, an array is nothing more than a block of memory set aside for a collection of a particular type

Creating an Array

- In Java: `int[] numbers = new int[100];`
- In Python: `numbers = []`
- In C:

```
int list[10];
double dlist[15];
```

- Note that here, the arrays **must** be dimensioned when they're declared
  - In older versions of C the dimension had to be a constant

Accessing Array Elements

- Retrieving the values of an array is the same as it is in Java:
  - `list[2] = 7;`
  - `list[1] = list[2] + 3;`
- However, one interesting note about C is there is no boundary checking, so:

```
list[10] = 7;
dlist[17] = 2.0;
```

…may work just fine.

Experiment: When will it Break?

- We can try modifying out-of-bounds array elements
- We can even do it in a loop to test the limits
  - Different operating systems / architectures may react differently
  - Let's demo this now…

- At this point, you might be wondering:
  - What is wrong with C?!
  - What is the meaning of life?

Accessing Array Values

- So we **can** do things like this in C:

```
int list[5];
list[10] = 7;
```

- Your program may work fine… or crash!
- It's **never** a good idea to do this
- So why does C let us do it anyway?

Safety and Performance

- C favors performance over safety
  - Compare: C program vs Python equivalent
  - Helpful: **time** command
- Especially in the glory days of C, adding lots of extra checks meant poor performance
  - Extra ‘if’ statement for each array access, etc…
- Sometimes these safety features aren’t necessary
  - Especially for perfect programmers?
- You can always implement safety features yourself

Creating an Array

- Let's create our list of integers:
  - `int list[10];`
- When we do this, C sets aside a place in memory for the array
  - It **doesn't** clear the memory unless we ask it to
  - A common cause of subtle bugs
- Creating a list of integers initialized to zero:
  - `int list[10] = { 0 };`

Memory Access

- What happens when you retrieve the value of `list[5]`?
    1. Find the location of list in memory
    2. Move to the proper offset:
       5 * 4 = byte 20
    3. Access the value
- Accessing `list[500]` is just moving to a position in memory and retrieving whatever is there

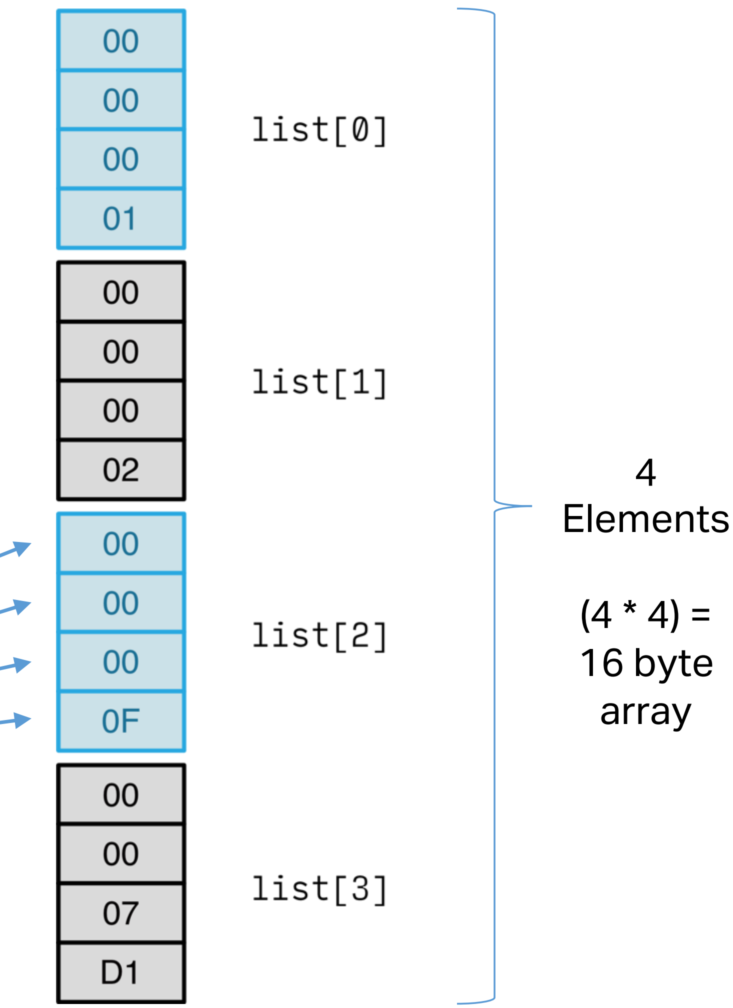
Visualizing Arrays in Memory

```c
/* Note: can calculate array
 * dimensions automatically! */

int list[] = {
    0,
    1,
    15,
    2001
};

sizeof(int) = 4
```

(4 bytes)



The **sizeof** operator

- We can use the **sizeof** operator in C to determine how big things are
  - **Somewhat** like:
    - len() in python
    - .length in Java, or
    - .size() in Java
- Much more low-level

```
size_t sz = sizeof(int);
printf("%zd\n", sz);
// Prints 4 (on my machine)
```

Array Size (1/2)

- Let’s try this out:

```c
int list[10];
size_t list_sz = sizeof(list);
```

- Any guesses on the output?
- On my machine, it’s 40:
  - 40 bytes (10 integers at 4 bytes each)
  - This can be different depending on architecture
- In C, `sizeof(char)` is guaranteed to be 1.

Array Size (2/2)

- Knowing the number of bytes in the array can be useful, but not **that** useful
- Usually we want to know how many elements there are in an array
- To do this, we'll divide by the array **type** (int - 4 bytes):

```
int list[10];
size_t list_sz =
     sizeof(list) / sizeof(list[0]);
printf("%zd\n", list_sz);
/* Prints 10 (on my machine) */
```

Behind the Scenes

- Arrays in C are actually pointers

  `int list[5];`

  `list` is the same as `&list[0];`

- You can't change what they point at, but otherwise they work the same

- So accessing `list[2]` is really just dereferencing a pointer that points **two** memory addresses from the start of the array
  - ...ever think about why we used 0-based arrays in CS?

We can make this more “fun...”

- Since arrays are just constant pointers, we have another way to access them:

```
list[5]
```

  Is the same thing as:

```
*(list + 5)
```

- Workflow:
  1. Locate the start of the array
  2. Move up **5** memory locations (4 bytes each*)
  3. Dereference the pointer to get our value

Pointer Arithmetic

- Manipulating pointers in this way is called **pointer arithmetic**

- `arr[i];`

  is the same as

  `*(arr + i);`

- `arr[6] = 42;`

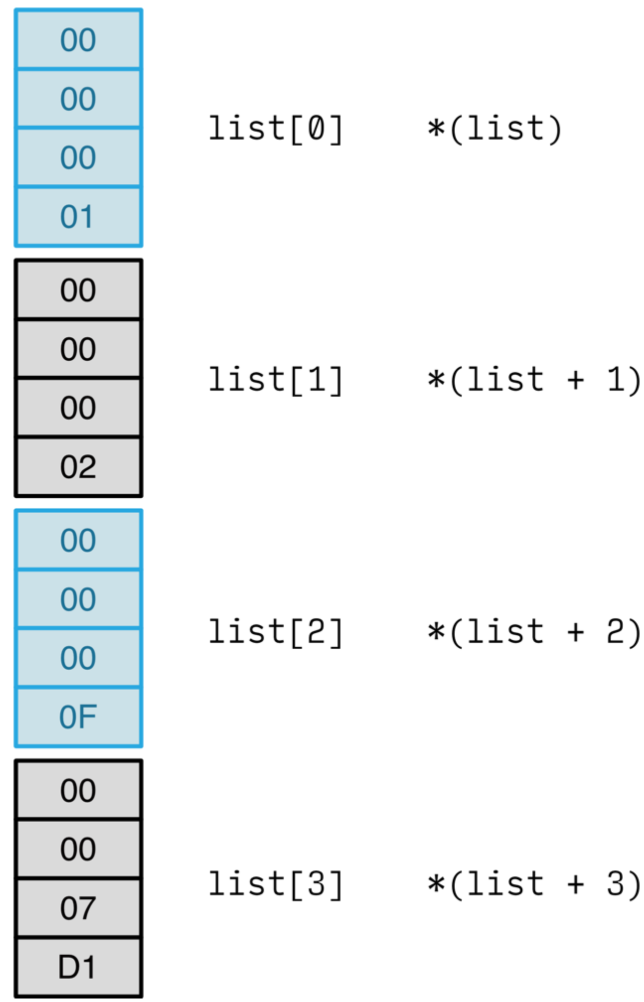
  is the same as

  `*(arr + 6) = 42;`

Visualizing Arrays

```
int list[] = {
    0,
    1,
    15,
    2001
};

sizeof(int) = 4
```



A Note on Pointer Arithmetic

- In general, stick with using regular array syntax
- You may see pointer arithmetic in production code, but it should only be used in situations that make the code **more** understandable
- Haphazardly showing off your knowledge of pointer arithmetic is a recipe for confusing code

Arrays as Function Arguments

- When we pass an array to a function, its pointer-based underpinnings begin to show
- If we modify an array element inside a function, will the change be reflected in the calling function?
  - Why?
- In fact, when an array is passed to a function it **decays** to a pointer
  - The function just receives a pointer to the first element in the array. That's it!

Array Decay

- When an array decays to a pointer, we lose some information
  - **Type** and **dimension**
- Let’s imagine someone just gives us a pointer
  - Do we know if it points to a single value?
  - Is it the start of an array?
- Functions are in the same situation: they don’t know where this pointer came from or where it’s been
  - **sizeof** doesn’t work as expected

Avoiding Decay

```
decay.c:22:19: warning: sizeof on array function
parameter will return size of 'int *' instead of
      'int []' [-Wsizeof-array-argument]
          sizeof(list),
```

- To avoid this situation, we need to pass in the size of the array as well.
- You may have wondered why the sizes of arrays are always being passed around in C code
  - This is why!

Homework 1: Arrays

- Let’s get started on HW1 (posted on the course website)
- This homework introduces you to a few new functions and gives you a chance to play with arrays