Tutorial 7

- Arrays.
- Pointer arithmetic.
- Midterm Review (TIME PERMITTING)

Arrays

Arrays can be thought of as pointers to a block of memory. They can be used to store a **fixed number** of elements of the **same type**.

Example of array syntax:

```
int my_array[3] = { 1, 2, 3 };
int x = my_array[0]; // x = 1
```

In the above code, `my_array` is a pointer to the first element of the array. The `[i]` syntax **dereferences** the $(i+1)^{\text{th}}$ element of the array.

> The syntax `a[i]` is shorthand for the equivalent expression `*(a+i)`.

Array Initialization

There are several ways to define an array:

```
int a[3]; // array is not fully defined

int b[3] = { 1, 2, 3 }; // array length is explicit

int c[] = { 1, 2, 3 }; // array length is inferred
// c has length 3, and we cannot change it later

int d[3] = {0}; // array of length 3, filled with zeros

int e[8] = { 7, 4, 1 };
// {7, 4, 1, 0, 0, 0, 0, 0}
```

Array Exercise

```
//See reverse.h for general documentation.
void reverse(int arr[], int len);
```

Pointer Arithmetic

Certain arithmetic operations can be performed on pointers. An integer can be **added or subtracted** to a pointer, and pointers of the same type can be **subtracted** from one another.

```
int a[10];
int *p = a;      // a is a pointer to first element
int *q = &a[9]; // address of 10th element
q = a + 9;       // equivalent
a[2] = q - p;    // set the value of 3rd element as 9
q = p + 1;       // now q == &a[1]
```

Addition of pointers is not allowed.

Array Notation vs Pointer Notation

If you ask the right people, you can start heated debates on whether to use `int a[]` or `int *a` in a parameter declaration.

They are equivalent.

Some people will argue for the first style because it is clear that the function takes an array.

Others believe the first style is misleading, as the array isn't actually passed by value and using `sizeof` on it is unintuitive.

You should be familiar with both ways of passing arrays.

Array Exercise 2, Electric Boogaloo

Write Reverse again, now using pointer arithmetic

(hint, this code will be essentially identical to reverse)

```
//See reverse.h for general documentation.
void reverse(int arr[], int len);
```

Midterm Review

- Practice material from the Winter 2014 midterm.
- Will cover some (but not all) of the questions.
- Midterm content: Everything up to / including modules.

Short Answer

List three advantages of modularization, and briefly explain.

Short Answer

List three advantages of modularization, and briefly explain:

- Re-usability: Build programs faster, and build large programs more easily.
- Maintainability: Makes debugging and changing a program easier.
- Abstraction: No need to know / understand all parts to use them. Allows changing internals without breaking things.

Clicker Question

For positive integers, the C modulo operator (%) behaves the same as the Racket quotient function.

A True.

B False.

C Which one is quotient again?

Clicker Question

For positive integers, the C modulo operator (%) behaves the same as the Racket quotient function.

A True.

B *False.

C Which one is quotient again?

The modulo operator behaves the same as the remainder function.

Clicker Question

Every C module must have a main function.

A True.

B False.

C True, but only on Tuesdays.

Clicker Question

Every C module must have a main function.

A True.

B *False.

C True, but only on Tuesdays.

A module does not require a main function. (Example: The stack module from the last assignment).

Short Answer

Racket uses *dynamic typing*. What kind of typing does C use? Give one advantage of the typing C uses. Provide a brief Racket example that demonstrates dynamic typing that isn't possible in C.

Short Answer

Racket uses *dynamic typing*. What kind of typing does C use? Give one advantage of the typing C uses. Provide a brief Racket example that demonstrates dynamic typing that isn't possible in C:

- C uses static typing.
- Advantage: Built-in contract, can detect type errors at compile-time, fast (no run-time type checking).
- Example (there are many):
  (define dyntype (if (>= x 0) x "invalid"))

Clicker Question

In C, `(a != 0) && (b/a == 2)` will produce an error if a is 0.

A True.

B False.

C Impossible to know without trying it.

Clicker Question

In C, `(a != 0) && (b/a == 2)` will produce an error if a is 0.

A True.

B *False.

C Impossible to know without trying it.

Because && is used, the first condition will short circuit the evaluation, preventing `b/a` from being executed.

Short Answer

Briefly explain the purpose of *requires* and *effects* function documentation in C.

Short Answer

Briefly explain the purpose of *requires* and *effects* function documentation in C:

- Requires: Identifies conditions that must be true when calling the function (e.g. restrictions on parameters).
- Effect: Identifies what a function prints, reads, or mutates.

Clicker Question

`printf("hello!\n")` is a C expression with a value of 7.

A True.

B False.

C The only “expressions” in C are my cries of pain when I’m forced to code in it.

Clicker Question

`printf("hello!\n")` is a C expression with a value of 7.

A *True.

B False.

C The only “expressions” in C are my cries of pain when I’m forced to code in it.

The return value of `printf` is the number of characters printed.

Short Answer

Write the declaration for a C function add that takes two ints, x and y, and returns an int x+y.

Short Answer

Write the declaration for a C function add that takes two ints, x and y, and returns an int x+y:

- No actual addition required!
- `int add(int x, int y);`

Clicker Question

In the following C code, the assignment operator appears only once:

```
bool nisfive = (n == 5);
```

A True.

B False.

C This has to be harder than it looks but I'm not seeing the trick.

Clicker Question

In the following C code, the assignment operator appears only once:

```
bool nisfive = (n == 5);
```

A True.

B *False.

C *This has to be harder than it looks but I'm not seeing the trick.

The code above uses = for initializaiton, NOT assignment. They are very slightly different (e.g. some struct-related notation can only be used during initialization, and not assignment).

Stack Trace

Draw the call stack immediately after “In exchange: ” is printed. For pointers, draw an arrow to the variable they point at.

Stack Trace

```
1  void exchange(int *pa, int* pb) {
2    int temp;
3    temp = *pa;
4    *pa = *pb;
5    *pb = temp;
6    printf("In exchange: ");
7        printf("a = %d, b = %d\n", *pa, *pb);
8  }
9
10 int main(void) {
11   int a, b;
12   a = 5;
13   b = 7;
14   printf("In main: ");
15   printf("a = %d, b = %d\n", a, b);
16   exchange(&a, &b);
17   printf("a = %d, b = %d\n", a, b);
18 }
```

Stack Trace

```
exchange:
  pa: [arrow pointing to a]
  pb: [arrow pointing to b]
  temp: 5
  return addr: main : 16
==========
main:
  a: 7
  b: 5
  return addr: OS
```