

# Course Recap

2024 Winter APS 105: Computer Fundamentals

Jon Eyolfson

Lecture 36

1.0.1

## We Need to Use a Program to Create Programs

A **compiler** transforms your source code into a program your OS can run



**Machine code** is the binary representation of instructions the CPU can run

We often use **compile** as a verb: "Let's compile our program."

## We Can Do Assignment as Part of a Variable Declaration

The full syntax for variable declaration is:

```
<type> <name> = <value>;
```

This statement creates a variable and assigns an **initial value**  
(an initial value is the first value the variable has, and it's optional)

We also call this **initialization**

## Types Used in This Course

`int`

`double`

`char`

`bool` (requires `#include <stdbool.h>`)

`void` (nothing)

Pointers: `<type> *`

Arrays: `<type> []`

## Input/Output in a Nutshell

### printf

Outputs the string to your terminal, use format specifiers to print values of variables

### scanf

Reads input from the terminal, use format specifiers to change values at addresses

Format specifiers:

%lf for `double`

%d for `int`


%c for `char`

%p for pointers (not needed for exam)

## Precedence Rules for Common C Operators

Operator	Associativity
++ -- (postfix)	Left-to-right
++ -- (prefix) (<type>) (cast) & (address-of) sizeof	Right-to-left
* / %	Left-to-right
+ -	Left-to-right
= += -= *= /= %= (assignments)	Right-to-left

Higher Precedence



Lower Precedence

## All Math Functions We Explored in the Course

```
double sqrt(double x); // Computes the square root of x
double pow(double x, double y); // Computes x to the power of y
double sin(double radians); // Computes sin using radians
M_PI; // Value of pi
double log(double x); // Computes the natural logarithm (to base e) of x
double log10(double x); // Computes the common logarithm (to base 10) of x
double fmod(double x, double y); // Computes the remainder of x divided by y
double fmin(double a, double b); // Outputs the smaller of the two values
double fmax(double a, double b); // Outputs the larger of the two values
double rint(double x); // Computes the nearest integer, rounding towards even
double ceil(double x); // Computes the first integer larger or equal to x
double floor(double x); // Computes the first integer smaller or equal to x
double trunc(double x); // Truncates x to an integer
int rand(void); // Returns a "random" number between 0 and RAND_MAX (inclusive)
void srand(unsigned int seed); // Changes the seed value that rand uses
```

## We Used If Statements to Conditionally Run Code

The syntax of an if statement is:

```
if (<expr>) <statement>  
else <statement>
```

However, you should **always** write it like:

```
if (<expression>) {  
    <statements>  
}  
else {  
    <statements>  
}
```

The expression evaluates to either true or false

Comparison operators: == != > >= < <=

Logical operators: ! && ||



## We Can Repeat Code When the Expression is True

The syntax of a while statement is:

```
while (<expr>) <stmt>
```

However, you should **always** write it like:

```
while (<expr>) {  
    <stmts>  
}
```

If we want the body of the loop to always execute once, we can do:

```
do {  
    <stmts>  
}  
while (<expr>);
```

## For Loops Usually Represent Bounded Repetition

The syntax of a for loop is:

```
for (<initialization stmt>; <conditional expr>; <increment expr>) <stmt>
```

However, you should **always** write it like:

```
for (<initialization stmt>; <conditional expr>; <increment expr>) {  
    <stmts>  
}
```

## We Can Write a Function Prototype for `addTwo` Before `main`

```
#include <stdio.h>
#include <stdlib.h>

int addTwo(int x);

int main(void) {
    printf("Result: %d\n", addTwo(4));
    return EXIT_SUCCESS;
}

int addTwo(int x) {
    return x + 2;
}
```

## The Scope is Part of the Program You Can Use a Variable

You can use a variable declaration within a { until the matching }

C declares function arguments, and **for** loop initializers in the next {

C is copy-by-value, so functions get a copy of the value for arguments

Beware of "shadowing" a variable (using the same name)

## **A Pointer is the Starting Address of a Value in Memory**

The & operator is the address of, its result is the pointer to the value  
For values that take up multiple bytes, it's always the lowest address

Each time we take the address of a variable, we add \* to its type

We can use the \* operator to access the value at an address,  
the type of the value is the type of the pointer with one \* removed

## Arrays in C are Multiple Values of the Same Type

The syntax to create an array is:

```
<type> <name>[<array_size>] = {<comma_separated_values>;
```

We can omit the <array\_size>, and

C will create an array with a length equal to the number of values

We can also omit the {<comma\_separated\_values>}

## We Can Use Pointer Arithmetic for Arrays

Assume we have our `int` array called `grades`:

`grades + 1` is a pointer to the second element

We can dereference the pointer, `*(grades + 1)`, to access the second value

The syntax to access an array element is just for your convenience

`grades[index]` is the same as `*(grades + index)`

If you do arithmetic between two pointer values,  
the result is the number of values between them

## We Can Request Memory Using `malloc`

Its function prototype in the C standard library is:

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

`size_t` is basically a positive integer type  
(the `sizeof(size_t)` depends on your machine)

The size argument is how many contiguous bytes to allocate

`malloc` returns a pointer to a starting address,  
you may then use size contiguous bytes



## Use Dynamic Memory Only When Needed

Dynamic memory is tricky to get correct, you need to:

- Remember to `free` when you're done using the memory
- Don't try to use the memory after you `free` (use-after-free)
- Don't call `free` twice on the same pointer (double free)

You should only use it when:

- Your function needs to return a pointer to valid memory
- You do not know the amount of memory you need at compile-time

## We Can Create 2D Arrays

The syntax for declaring 2D arrays is:

```
<type> <name>[<first_size>][<second_size>;
```

Where you replace:

<type> by the type for each value (or **element**) of the array

<name> by a name you want to give the array (group of values)

<first\_size> by the number of arrays you want in the next dimension

<second\_size> by the number of elements in each array

## The Rest of a Row Gets Filled with 0s

We can initialize a 2D array like so:

```
int table[][3] = {  
    {1, 2},  
    {4, 5}  
};
```

If we output we'll see:

```
table[0][0]: 1  
table[0][1]: 2  
table[0][2]: 0  
table[1][0]: 4  
table[1][1]: 5  
table[1][2]: 0
```

Note: this is the same for plain arrays, if you initialize at least one value, the rest of the **known** size is filled with 0s

## You Can Use Dynamic Lengths in Function Arguments

You could write a function prototype as:

```
void foo(int numRows, int numCols, int table[][numCols]);
```

However, the variable used in the multidimensional array must be declared before the array itself

You cannot write:

```
void foo(int numRows, int table[][numCols], int numCols);
```

You could optionally write:

```
void foo(int numRows, int numCols, int table[numRows][numCols]);
```

## The Previous Example Isn't a True Multidimensional Array

In true multidimensional arrays:

```
table[i][j]; is the same as table[i * NUM_COLS + j];
```

However, if you allocate memory dynamically, you do:

```
int *row = table[i];  
int element = row[j];
```

The declaration of table (if we used `malloc`) could be: `int *table[];`

However, we could also write it as: `int **table;`

Our first double pointer!

## Strings Use Memory, and are Difficult to Use Correctly

A string is an array of characters, in order  
to know a string ends, C adds a 0 byte (character `'\0'`) at the end

The format specifier (that you should only use for `printf`) is `%s`

You need to make sure there's enough memory to hold the string  
Buffer overflows are a serious security issue

For user input you should use either `fgets` or `getline`

## Summary of String Functions

```
#include <string.h>
size_t strlen(const char *s);
size_t strlen(const char *s, size_t maxlen);
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
char *strstr(const char *haystack, const char *needle);
char *strchr(const char *s, int c);

#include <stdlib.h>
int atoi(const char *s);
double atof(const char *s);
```

## **A Recursive Function Calls Itself**

We need two things:

1. a base case: a simple solution we know
2. a recursive step: reduces the problem to a smaller version of itself



## You Can Group Variables Within a Structure

You can create your own type with `struct`, its syntax is:

```
struct <name> {  
    <variable declarations>  
};
```

Where you replace:

<name> with the name you'd like to give the group of variables

<variable declarations> with as many variable declarations as you wish

You should define a `struct` just below the includes, and not within a function

## You Can Rename Types with `typedef`

The syntax of a `typedef`, is:

```
typedef <type> <new_name>;
```

Where you replace:

`<new_name>` by the name of whatever you'd like to name your type

`<type>` by the type you would like to use when you use `<new_name>`

## We Can Use `typedef` Is to Save Us from Writing `struct`

The style on the exam is usually:

```
typedef struct point {  
    double x;  
    double y;  
} Point;
```

Afterwards, you can create a variable with: `Point p1;`

You can access the fields of `p1` with `.`, e.g. `p1.x`

If `p1` is a pointer access the fields using `->`, e.g. `p1->x`

We can initialize structures like arrays, e.g. `Point p1 = {1.0, 2.0};`

C sets the values in order within the structure

## **A Linked List is Sequence of Nodes**

To represent a linked list, we only need to keep track of the first node  
Each node keeps track of the next node in the list

This allows some more flexibility over arrays in certain circumstances  
We can update pointers instead of changing where values are in memory

## All of the Linked List Functions We Wrote in One Day

```
void freeLinkedList(linked_list_t *linked_list);
void removeNode(linked_list_t *linked_list, node_t *node);
bool isEmpty(linked_list_t *linked_list);
int length(linked_list_t *linked_list);
void insertAfter(node_t *after, node_t *node);
void insertBefore(linked_list_t *linked_list, node_t *before, node_t *node);
void insertEnd(linked_list_t *linked_list, node_t *node);
node_t *insertBack(linked_list_t *linked_list, int val);
node_t *removeFront(linked_list_t *linked_list);
```

After, we removed nodes with matching values, you should practice!

## **Sorting is One of the Most Common Problems**

Computers are very fast, and they don't care about the order of the data

However, programs almost always sort results to make it presentable

We discussed several sorting algorithms, that you should know:

- Bubble sort

- Selection sort

- Insertion sort (not covered in other sections)

## **We Can Search a Sorted Array Faster**

If we have an unsorted array, we need to use linear search

However, if we search a sorted array, we can use binary search

## What's Next?

### Second Year

ECE243: Computer Organization

Design of central processing unit, understand hardware

ECE244: Programming Fundamentals

C++, big-O complexity analysis, and testing and debugging

ECE297: Software Design and Communication

Work on a larger software design project

### Third Year

ECE344: Operating Systems

Explore the software between your program and hardware

ECE345: Algorithms and Data Structures

Expand on ECE244 with more advanced concepts