# CS 240: Programming in C

Lecture 19: Dynamic Arrays, Types, Preprocessor

PURDUE UNIVERSITY

# Announcements

- Homework 8 due Friday
  - 10 pts extra credit if you turn it in by tonight!
- Homework 9 is also out

# Dynamic 2D arrays

- We've seen how to dynamically allocate memory for 1-dimensional arrays

```
int n;
scanf("%d", &n);
int *arr = malloc(n * sizeof(int));
for (int i = 0; i < n; i++) {
  arr[i] = 3 * i - 1;
}
```

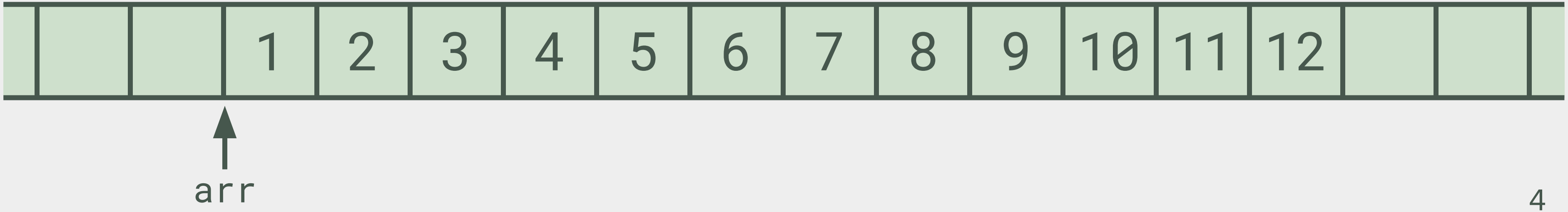
- How can we do this for 2D arrays?

# 2D arrays on the stack

- How does the following appear in memory?

```
int arr[3][4] = { { 1,  2,  3,  4 },
                  { 5,  6,  7,  8 },
                  { 9, 10, 11, 12 } };
```

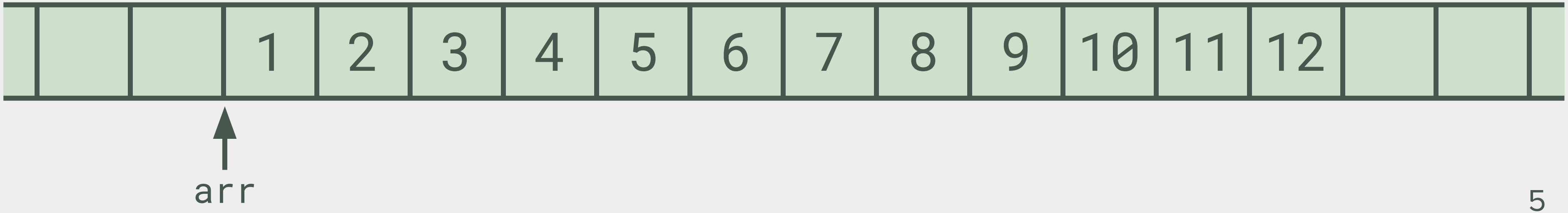
- It is placed in memory by each column in the first row, then each column in the second row, etc.
- “Row-major order”



# *2D arrays on the stack*

- How can we access row i, column j?
- Which of these work? All are equivalent!

```
arr[i][j];
*(arr[i] + j)
(*(arr + i))[j]
*((*(arr + i)) + j)
*(&arr[0][0] + 4 * i + j)
```



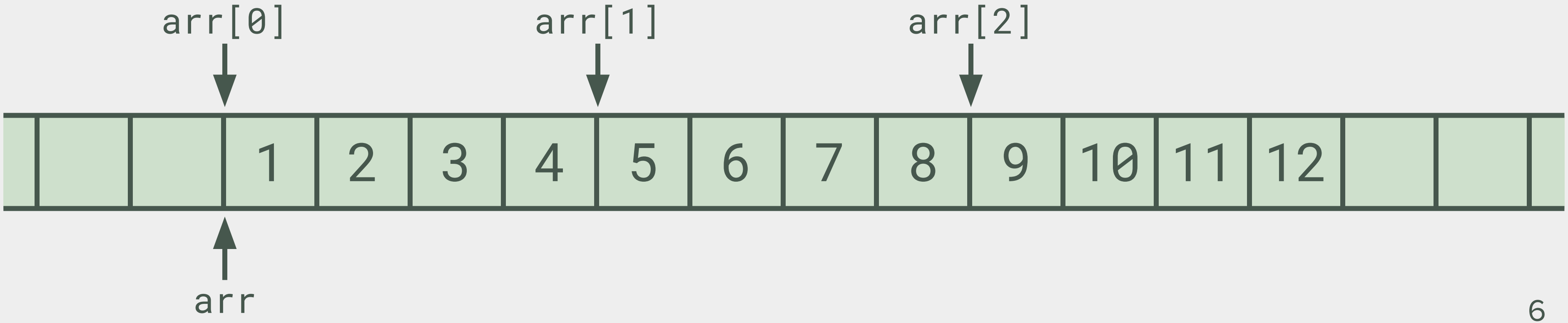
# *2D arrays on the stack*

```
arr[i][j];
*(arr[i] + j)
(*(arr + i))[j]
*((*(arr + i)) + j)
*(&arr[0][0] + 4 * i + j)  /* 4 == # columns */
```

- What is the type of `(arr + i)`?

```
int (*)[4]
```

"pointer to array of 4 ints"



# 2D arrays on the heap

- “Dynamic arrays” == heap-allocated
- We can only use dynamic arrays if we know the amount of memory needed before allocation
- BUT, we don’t need to know it at compile-time
  - If we know at compile-time, we can allocate on the stack
  - Provided it’s not too large
- If we can’t know the size before creation, we must use linked lists or another dynamic data structure

# *Dynamic 2D array creation*

- Recall that 2D arrays are contiguous in memory
- We can think of them as 1D arrays
- Simply allocate (rows x cols) elements

```
int *arr = calloc(rows * cols, sizeof(int));
assert(arr);
```

# Dynamic 2D array access

- Can we access row i, column j in the same way?

```
arr[i][j]
etc...
```

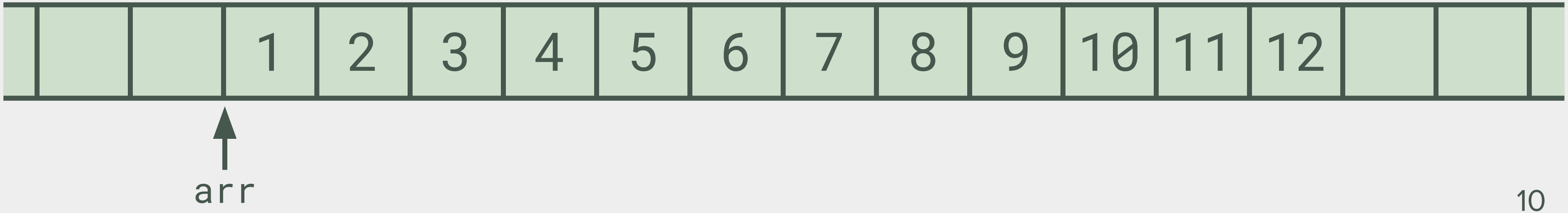
# Dynamic 2D array access

- Can we access row i, column j in the same way?

```
arr[i][j]
etc...
```

- No, because array is just a pointer to int
- Instead we need to calculate the index

`*(arr + cols * i + j)` or `arr[cols * i + j]`



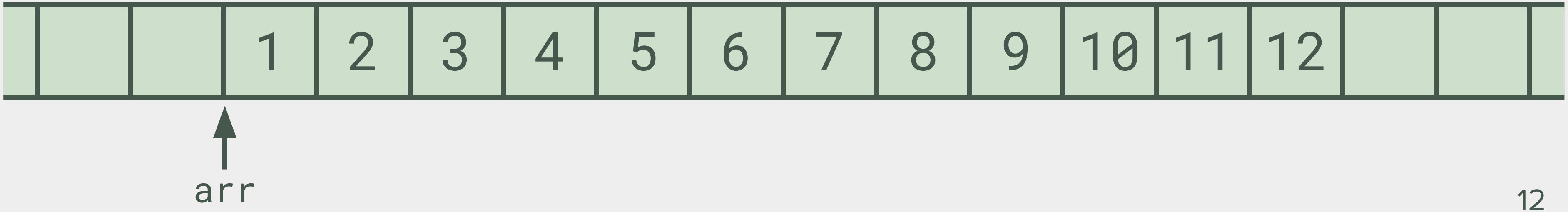
# *Dynamic 2D notes*

- Access is only truly valid if
  - $0 \leq i < rows$ and $0 \leq j < cols$
- What happens outside this range?

# Dynamic 2D notes

- Access is only truly valid if
  - 0 ≤ i < rows and 0 ≤ j < cols
- What happens outside this range?
  - rows = 3, cols = 4
  - i = 1, j = 6

```
*(arr + cols * i + j)
```



# 3D arrays

- We can also make 3D arrays!

```
int *arr = calloc(width * height * depth, sizeof(int));
assert(arr);
```

- How do we index into it?

```
int x, y, z;
arr[ ??? ]
```

# 3D arrays

- We can also make 3D arrays!

```
int *arr = calloc(width * height * depth, sizeof(int));
assert(arr);
```

- How do we index into it?

```
int x, y, z;
arr[width * height * z + width * y + x]
```

- Assumes the order of dimensions in memory is Z,Y,X
  - x is the “most frequently changing coordinate”
  - z is the “least frequently changing coordinate”
- The order is ultimately up to us to decide

# *Types*

- We are already familiar with the many basic types in the C language. They are called **first class** types:
    - char usually 1 byte
    - short usually 2 bytes
    - int usually 4 bytes
    - long usually 8 bytes, sometimes 4 bytes
    - long long usually 8 bytes
    - float usually 4 bytes
    - double usually 8 bytes

# *Sizes of data*

- There are general rules-of-thumb for the size of a variable, depending on type.
    - The only way to be sure is to use sizeof
- There are rules that say, for instance, that an **int** must be no smaller than a **short** and no larger than a **long**
- Types are automatically **promoted** to the next larger type of the same family (e.g. integer or floating-point) within arithmetic operations

# Conversion

- After promotion, arguments to an operator are checked
  - If the same, proceed
  - Otherwise, conversions may take place
  - For each type, if one of the arguments is that type, the other is converted to the same type, in order:
    - long double, double, float, unsigned long, long, unsigned, int

# *Type modifiers*

- Integer types (char, short, int, long, long long) can have an additional modifier to indicate to the compiler whether the datum represents a signed or unsigned (always non-negative) value.

```
unsigned char x = 200;  /* OK */
signed char y = 200;    /* overflow */
```

- For non-integer types, “signed” has no meaning
- The default modifier for **int** is signed
- What’s the default modifier for **char**?

# Second-class types

- Constructed types are second-class types. They are created by the programmer.
- Examples of derived types include anything that the programmer declares that is a struct, union, enum, or pointer to anything

# Assignments

- You can make assignments between compatible types or types that can be promoted. This usually works between all first-class types

```
int i;
unsigned int ui;
float f;
char c;

f = ui;
i = f;
c = f;
```

# Bad assignments

- You cannot make assignments between data of differing second-class types

```
struct my_struct {
  int x;
} str1;

struct your_struct {
  int x;
} str2;

str2 = str1;   /* not allowed */
```

# *Type qualifiers*

- There are two type qualifiers that can be used with any type declaration:
  - **const**: this datum must not be modified
  - **volatile**: this datum may be modified by something outside the program! (e.g. the hardware, another program, multi-threaded programs)
- Only one at a time may be used for any single declaration

# *Type qualifier example*

```
const double PI = 3.14159265358979323846264;

int get_factor() {
  const int factor = 45;
  factor--;
  return factor;
}
```

```
error: decrement of read-only variable ‘factor’
```

PURDUE UNIVERSITY

# Why const?

- Why not use #define?
- const creates a variable; #define just replaces text
  - This means you get the benefit of type-checking with const
  - const variables also have scope (can be local to a function)

# *const pointers*

- The const keyword can be used with pointer declarations in interesting ways:

```
const int *ptr;
```

  - Means that ptr points to an integer whose value cannot be modified

```
int * const ptr;
```

  - Means that ptr is an unmodifiable pointer that points to an integer whose value *can* be modified

```
const int * const ptr;
```

# *const pointer arguments*

- Here is the actual prototype for strcpy():

```
char *strcpy(char *dest, const char *src);
```

    - This is a **guarantee** made by the author of strcpy() that the string passed through the src argument will not be modified.
    - The string that is passed through src does not need to be **defined** as const
- Anytime you create a function whose arguments accept a pointer whose dereferenced values will not be modified, those arguments should be declared as const.

# const pointer examples

```
unsigned int count_tree_nodes(const tree_t *root) {
  if (root == NULL)
    return 0;
  return 1 + count_tree_nodes(root->left)
           + count_tree_nodes(root->right);
}
```

```
unsigned int strlen(const char *str) {
  unsigned int len = 0;
  while (*str++ != ‘\0’)
    len++;
  return len;
}
```

Are we modifying anything that str points to here?

# *Storage classes*

- There are two storage classes in the C language that we really care about. (There are more, but you'll rarely use them)
    - **extern**: The datum is defined in some other module
    - **static**: (If the datum is a local variable) the datum is initialized only once and retains its value between invocation of the function
    - **static**: (If the datum is a global variable) the datum is not visible from other modules
- Use either extern or static, but not both

# *When to use extern*

- If you are developing an application that has global variables whose values are accessed from multiple C files, each variable must be **defined** in only one C file and **declared as extern** in all other modules where the variable is referenced

# Two modules

module1.c

```
unsigned int counter;
double temp;
```

module2.c

```
extern unsigned int counter;
extern double temp;

void increment_count() {
  counter++;
}
```

# static local variables

- Consider a function that we want to use to generate and return a new serial number each time it's called…

```
unsigned int new_serial() {
  static unsigned int serial = 45000;
  return serial++;
}

int main() {
  printf(“First: %d\n”, new_serial());
  printf(“Second: %d\n”, new_serial());
  return 0;
}
```

PURDUE UNIVERSITY

# *static global variables*

- Use a static global when the variable must be visible to other functions in the same module but must **not** be seen (or called) from other modules that are linked into the application...

```
static int private_data;
void my_function() {
  private_data = 15;
}

static void increment_private() {
  private_data++;
}
```

# *Why do we use any of these?*

- You can get away with writing any program without any type qualifiers or storage classes (except extern)
- Using **static** improves software modularity and makes you less prone to violate an assumption that you may have made long ago (or many lines ago)
- Using **const** reminds you (or guarantees to a customer) that something should not be modified
- If you actually need to use **volatile** you'll usually know why...

# The preprocessor

- When a .c file is complied, it is first scanned and modified by a preprocessor before being handed to the real compiler
- If the preprocessor finds a line that begins with a #, it hides it from the compiler and makes a special note of it
    - Or, perhaps, takes other actions
- We’ve seen only two preprocessors directives so far:
    - #define and #include

# #include

- `#include` pulls a header file into another file

```
#include "file.h"
```

  - Pull in file.h from the **present directory**

```
#include <file.h>
```

  - Pull in **/usr/include/**file.h

PURDUE UNIVERSITY

# Example of #include

/home/may5/x.c

```
#include <stdio.h>
#include “x.h”

int main() {
    printf(“Val %d\n”, X);
    return 0;
}
```

/usr/include/stdio.h

```
/*
 * scary things
 * in this file...
 */
typedef FILE ...
```

/home/may5/x.h

```
#define X (3456)
```

PURDUE UNIVERSITY

# Example of #include

/home/may5/x.c

```
#include <stdio.h>
#include “x.h”

int main() {
    printf(“Val %d\n”, X);
    return 0;
}
```

/usr/include/stdio.h

```
/*
 * scary things
 * in this file...
 */
typedef FILE ...
```

/home/may5/x.h

```
#define X (3456)
```

# Final result of #include

```
/*
 * scary things
 * in this file...
 */
typedef FILE ...

#define X (3456)

int main() {
    printf(“Val %d\n”, X);
    return 0;
}
```

- All of the things that previously resided in separate files were pulled together into one stream
- **This** gets fed to the compiler

# More preprocessor directives

- An example:

```
#define TESTING

  x = some_function(y);
#ifdef TESTING
  printf("Debug point!\n");
  x = x + 5;
#else
  x = x + 5;
#endif
```

- If we turn off the TESTING definition, the debug statements are no longer compiled

# For next lecture

- Read 1.10, 2.1, 2.2, 2.4, 2.7, 4.4, 4.7, A4, A8 in K&R
  - Beej's Ch. 6 and 12.11

# Slides

- Slides are heavily based on Prof. Turkstra's material from previous semesters.