PURDUE UNIVERSITY®

CS 240: Programming in C

Lecture 17: Trees

Prof. Jeff Turkstra

Announcements

- Homework 8 due Wednesday, 3/26
- Midterm Exam 2 is on Thursday 4/10!

Reading

- (re-) read 5.7-5.9 in K&R
  - Beej's 12.6
- Read 1.10, 2.1, 2.2, 2.4, 2.7, 4.4, 4.6, 4.7, A4, A8.1 in K&R :-)
  - Beej's Ch. 6 and 12.11 (incomplete)

A note on debuggers

- As personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugging sessions are transient.
  - The Practice of Pogramming, Kernighan and Pike
- …or Linus Torvalds:
  - http://lwn.net/2000/0914/a/lt-debugger.php3

Other recursion examples

```
void countup(int n) {
  if (n >= 0) {
    countup(n-1);
    printf(“%d...\n”, n);
  }
  return;
}

int main() {
  countup(10);
  return 0;
}
```

Factorial

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n – 1);
}
```

Fibonacci sequence

```
/*
 * Compute one number in the
 * Fibonacci sequence:
 * 1 1 2 3 5 8 13 21 34 55 89 144…
 */
int fibonacci(int n) {
  if (n == 0)
    return 1;
  if (n == 1)
    return 1;

  return (fibonacci(n – 1) +
          fibonacci(n – 2));
}
```

When using recursion…

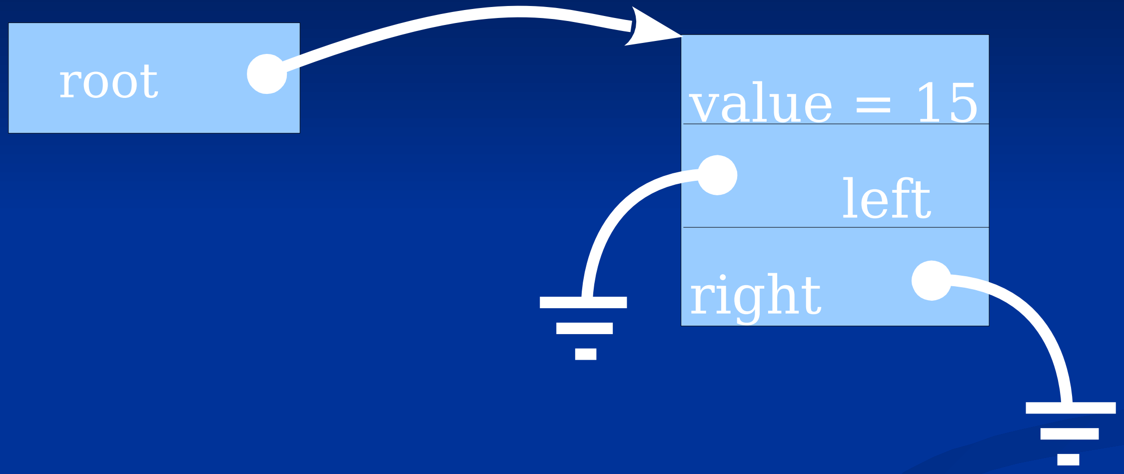
- You always need to tell the function when to stop invoking itself
- Don’t return a pointer to something on the stack
  - I.e. don’t return an address of a local variable
- Don’t recurse too deeply…
  - You will run out of stack space

Trees

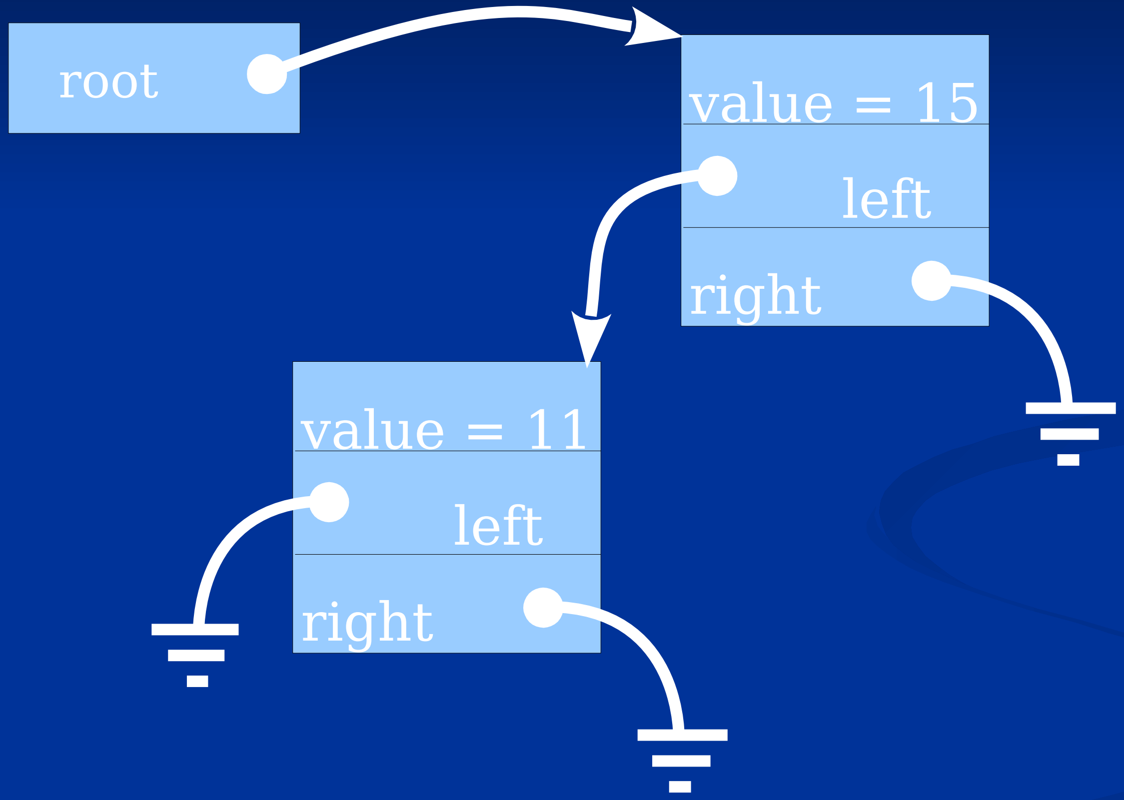
- Until now, we've only looked at lists that have one "dimension"
  - Forward/backward or next/previous
- Consider a structure that acts as a "parent" and has at most two "children" - a binary tree

```
struct node {
  int value;
  struct node *left;
  struct node *right;
};
```

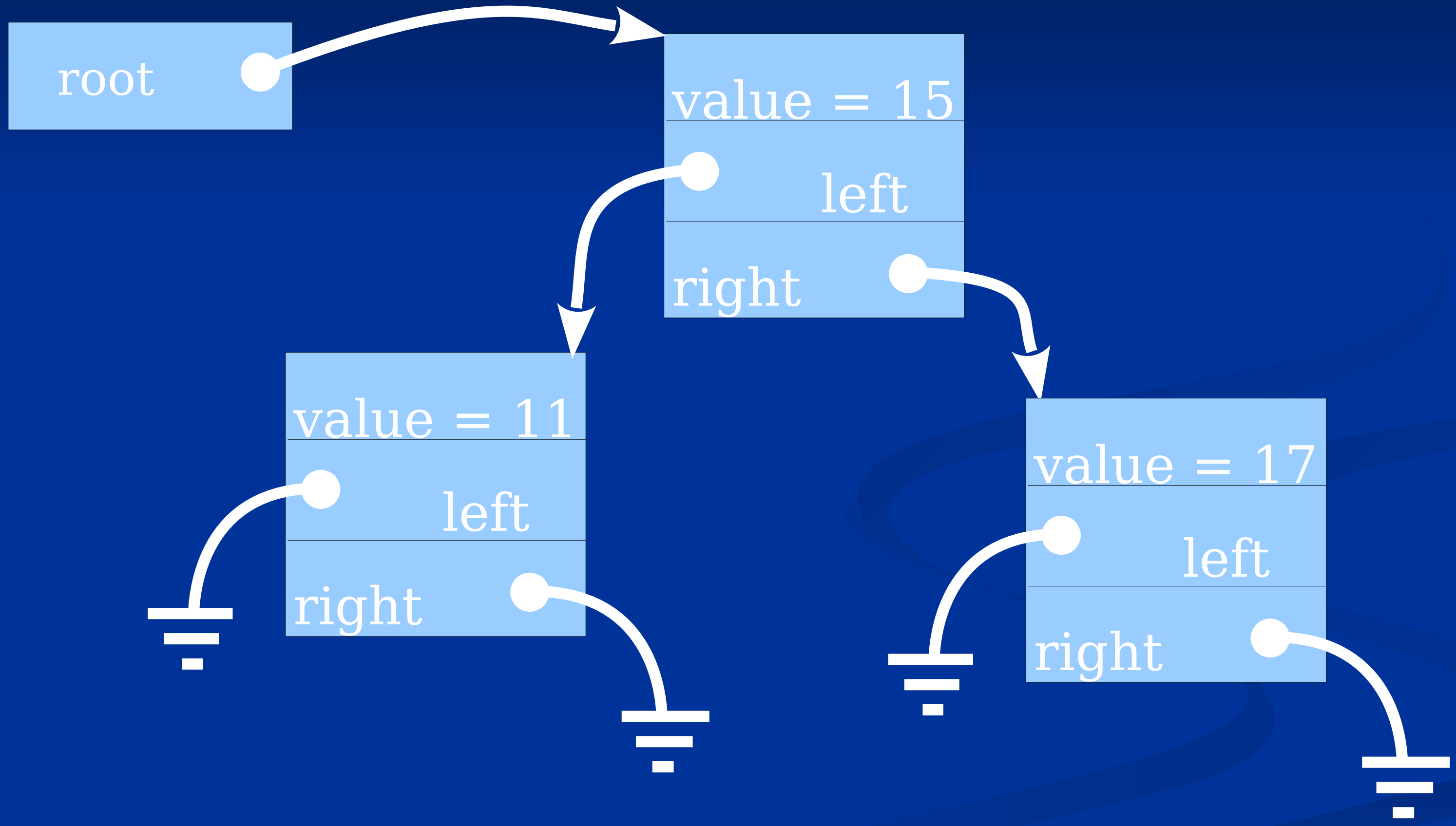
What does a tree look like – single node



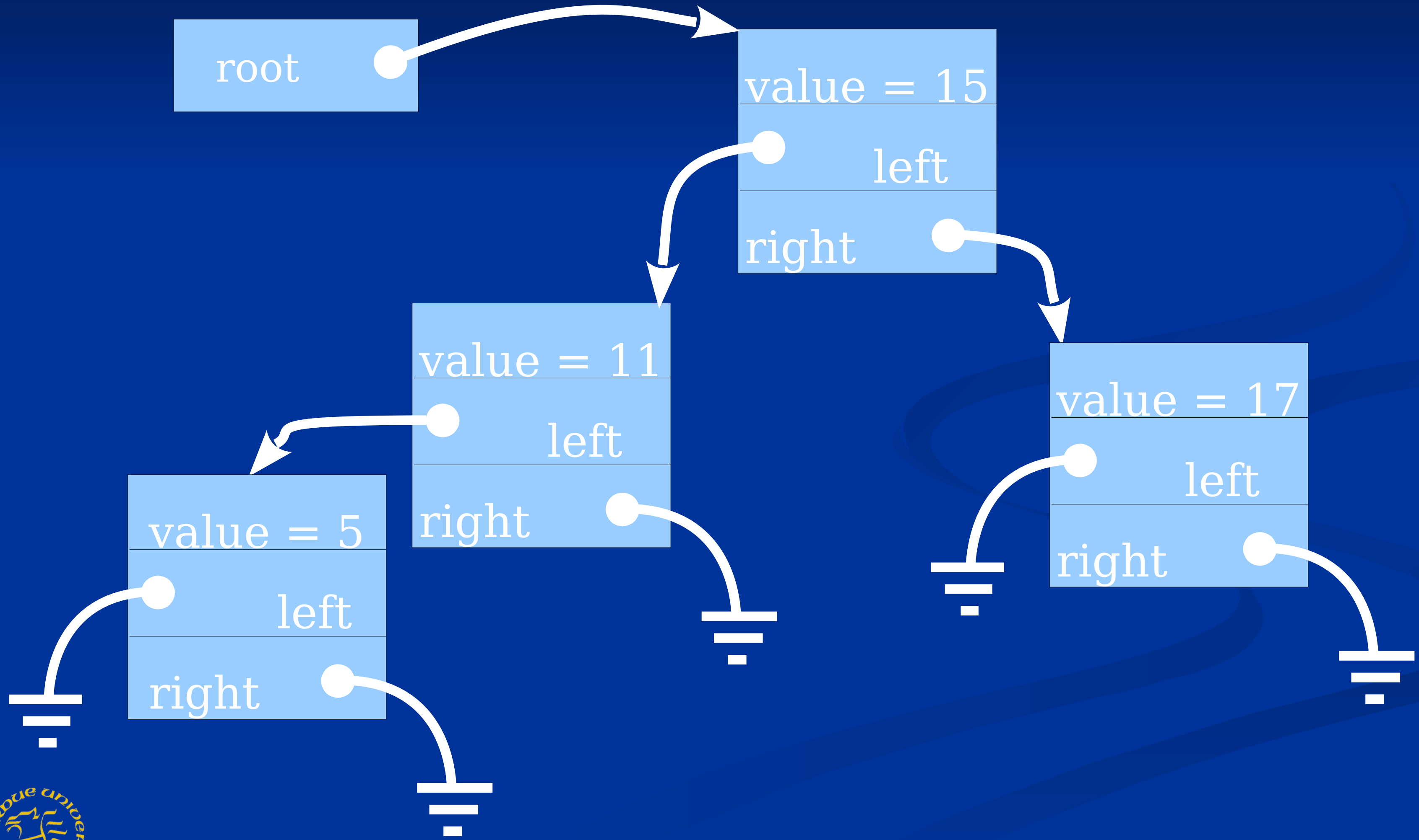
What does a tree look like – parent and left child



What does a tree look like – parent and two children



What does a tree look like – lots of children



Interesting properties of trees

- Trees are fun to use because you can easily add more children to the existing children
- With the trees we're working with:
  - The left child always has a value less than or equal to the parent's value
  - The right child always has a value greater than the parent's value
- You can always add a new child in the proper order
  - (to the left or right of the parent)
- The tree is always fully sorted
- The tree is easily searchable

Tree functions (create)

```
struct node *create_node(int value) {
  struct node *ptr = NULL;

  ptr = malloc(sizeof(struct node));
  assert(ptr != NULL);

  ptr->left = NULL;
  ptr->right = NULL;
  ptr->value = value;

  return ptr;
}
```

Tree functions (insert, iterative)

```
void insert_node(struct node *root, struct node *new) {
  while (1) {
    if (new->value <= root->value) {
      if (root->left == NULL) {
        root->left = new;
        return;
      }
      else {
        root = root->left;
      }
    }
    else {
      if (root->right == NULL) {
        root->right = new;
        return;
      }
      else {
        root = root->right;
      }
    }
  }
}
```

Tree functions (insert, recursive)

```
void insert_node(struct node *root, struct node *new) {
  if (new->value <= root->value) {
    if (root->left == NULL) {
      root->left = new;
      return;
    }
    else {
      insert_node(root->left, new);
    }
  }
  else {
    if (root->right == NULL) {
      root->right = new;
      return;
    }
    else {
      insert_node(root->right, new);
    }
  }
}
```

More about trees

- Searching an ordered binary tree is just as easy as inserting something in a tree:

  1. Set a pointer to point at the root structure

  2. If the value we’re looking for == the structure’s value, return the pointer

  3. If the search value is < the pointed to structure’s value, go left
     - E.g. pointer = pointer->left
     - And goto (2)

  4. Otherwise go right and goto (2)

  5. If the pointer is ever NULL,
     return NULL to indicate the value was not found

Recursive tree_find()

```
struct node *tree_find(struct node *root, int value) {
  if (root == NULL)
    return NULL;  /* Not found */

  if (value == root->value)
    return root;  /* Found it */

  if (value < root->value)  /* Go left */
    return tree_find(root->left, value);

  return tree_find(root->right, value);
}
```

Lecture Quiz 9

1. Tell me something fun that you did over spring break.

2. Any feedback about the course? Lectures? TAs? Homeworks? Exams? Anything else?

How do we get at the sorted content of a tree?

- We know that an ordered binary tree is fully sorted…
- The “least” element in the tree is at the far left
- The “greatest” element in the tree is at the far right
- Our tree nodes do not point back to their parents
  - How can we start at the far left and go through each node in order???

Tree traversal

- Accessing each of the nodes of a tree in order is often called tree traversal or iterating over a tree. We can do this in several ways:
  - Least to greatest – for each node, access the left node recursively, then the node itself, then the right node recursively:
    L-N-R
  - Greatest to least – Same way, except R-N-L
  - Prefix – N-L-R
  - Postfix – L-R-N

Example of ordered printing

```
void print_tree(struct node *ptr) {
  if (ptr == NULL)
    return;

  print_tree(ptr->left);  /* Go left */

  printf(“%d\n”, ptr->value);  /* Node */

  print_tree(ptr->right);  /* Go right */
}
```

Pointers to functions and trees

- Tree manipulation routines that we've looked at so far have assumed that one of the elements of the tree node was the key of the search/sort
- What if we had multiple items in the tree node structure and we wanted to be able to build trees by sorting against one of them?

```
struct node {
  struct node *left;
  struct node *right;
  char *name;
  char *title;
  char *phone;
  char *location;
}
```

Fields that we might sort by…

New tree_search()

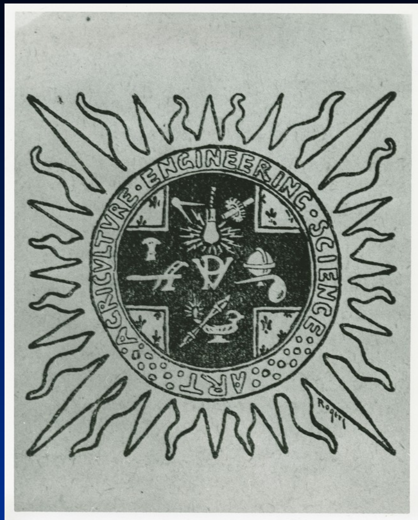
```
struct node *tree_search(
             int (*compare)(struct node *, char *),
             struct node *root, char *item)) {
  if (root == NULL)
    return NULL;
  if (compare(root, item) == 0)
    return root;
  if (compare(root, item) > 0)
    return tree_search(compare, root->left, item);
  return tree_search(compare, root->right, item);
}
```

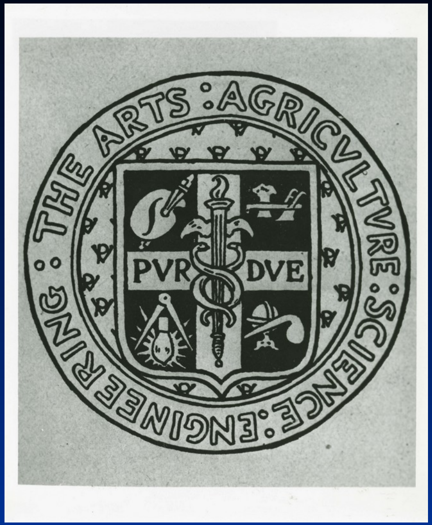
Example comparison function

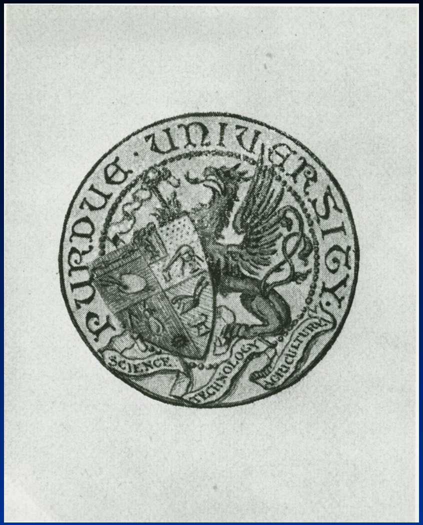
```
/*
 * Definition of comparison:
 * zero:     equal
 * negative: structure value ‘less than’ item
 * positive: structure value ‘greater than’ item
 */

int compare_name(struct node *ptr, char *item) {
  return strcmp(ptr->name, item);
}

/*
 * Example of calling tree_search()…
 */
ptr = tree_search(compare_name, root, “Jeff”);
```

Purdue Trivia

- The Purdue University seal has undergone nine iterations
  - First designed by Bruce Rogers in 1890
  - Most recent design by Al Gowan in 1968
  - The griffin symbolizes strength, with a three part shield reflecting Purdue's three permanent aims:
    - Education
    - Research
    - Service

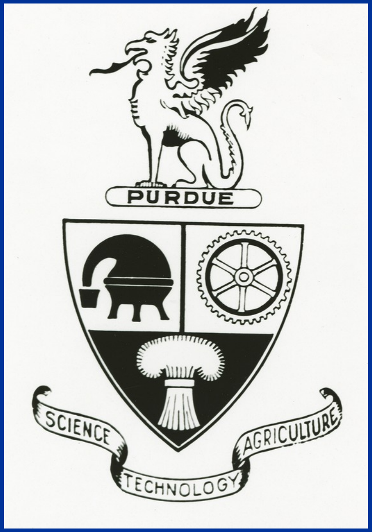

AGRICULTURE · ENGINEERING · SCIENCE · ART


THE ARTS : AGRICULTURE : SCIENCE : ENGINEERING :
PVR DVE

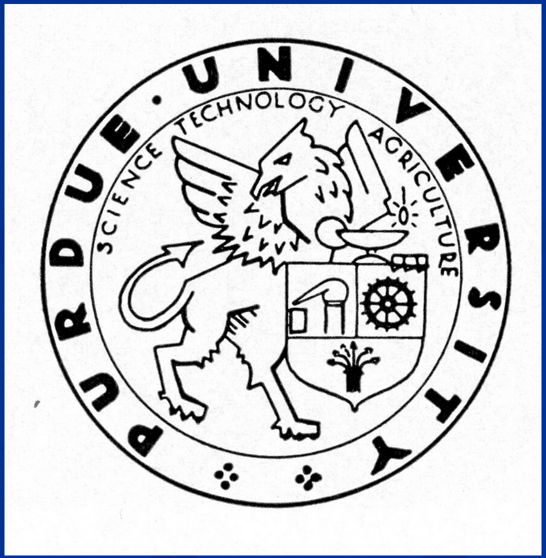

PURDUE · UNIVERSITY
SCIENCE
TECHNOLOGY
AGRICULTURE


PURDUE · UNIVERSITY
SCIENCE
TECHNOLOGY
AGRICULTURE


PURDUE UNIVERSITY
SCIENCE
TECHNOLOGY
AGRICULTURE


PURDUE
SCIENCE
TECHNOLOGY
AGRICULTURE


PURDUE · UNIVERSITY
SCIENCE
TECHNOLOGY
AGRICULTURE


PURDUE · UNIVERSITY
SCIENCE
TECHNOLOGY
AGRICULTURE

Dynamic 2D arrays

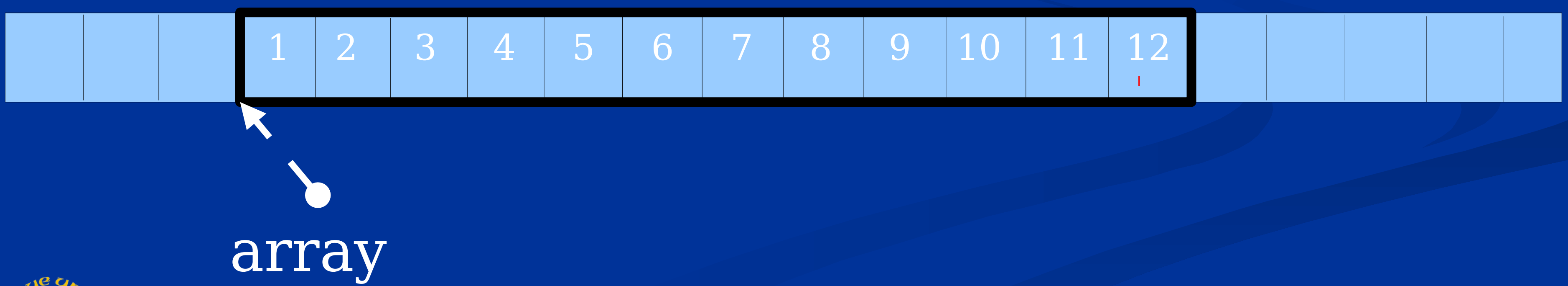
- Suppose we want to use a 2-dimensional array, but we do not initially know (at compile time) the size of the array. How can we do this?
- Recall that we dynamically allocate memory for 1-dimensional arrays (strings) without problems
- Also remember that all of memory (even for 2D, 3D, etc arrays) is still ultimately configured as a long, linear row of mailboxes. How does the computer access 2D arrays?

Arrangement of 2D arrays

- How does the following appear in memory?

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

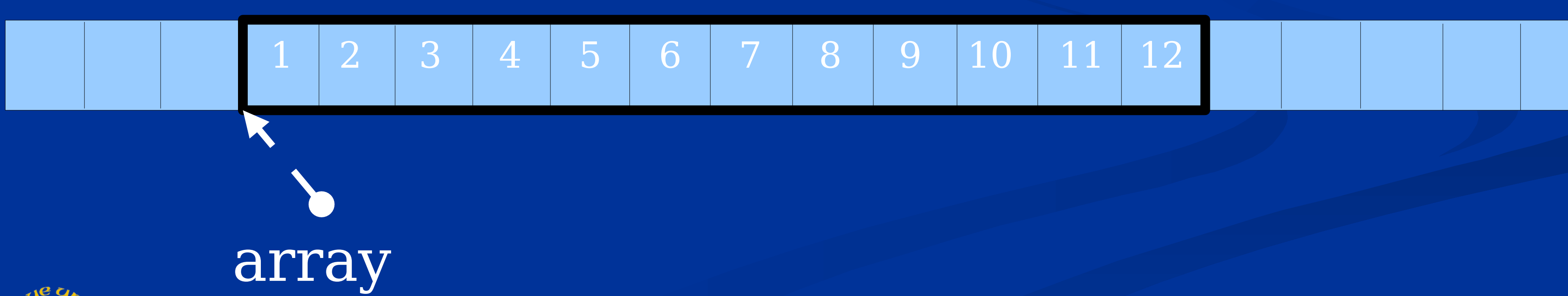
- It is placed in memory by each column in the first row, then each column in the second row, etc
- In other words:



Access of 2D elements

- How can one access row i, column j? Which of the following would work?

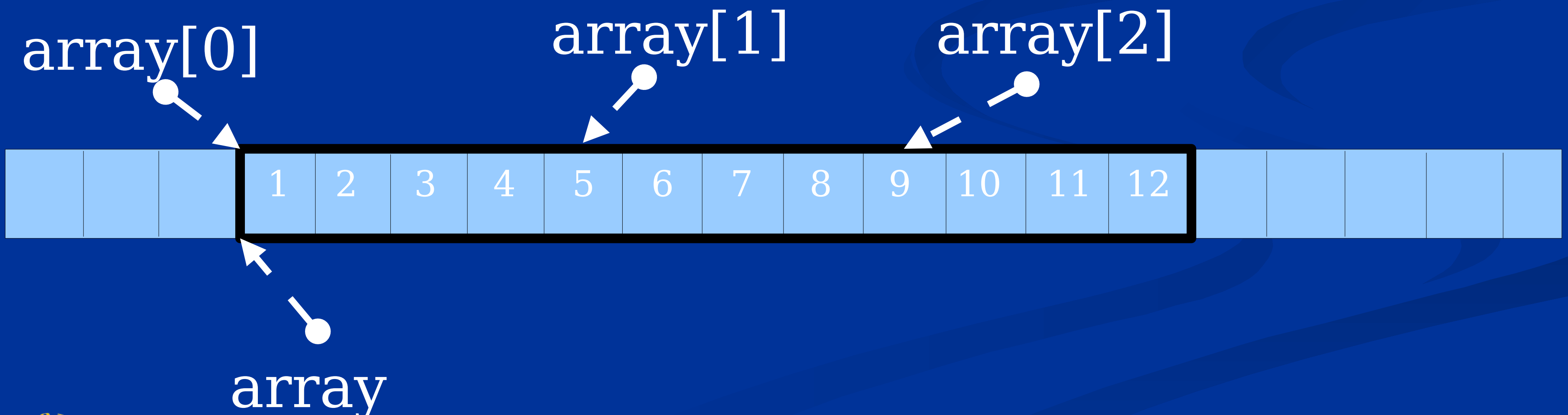
```
array[i][j]
*(array[i] + j)
(*(array + i))[j]
*((*(array + i)) + j)
*(&array[0][0] + 4 * i + j)
```



Access of 2D elements

- All are equivalent! Why?

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



Dynamic 2D creation

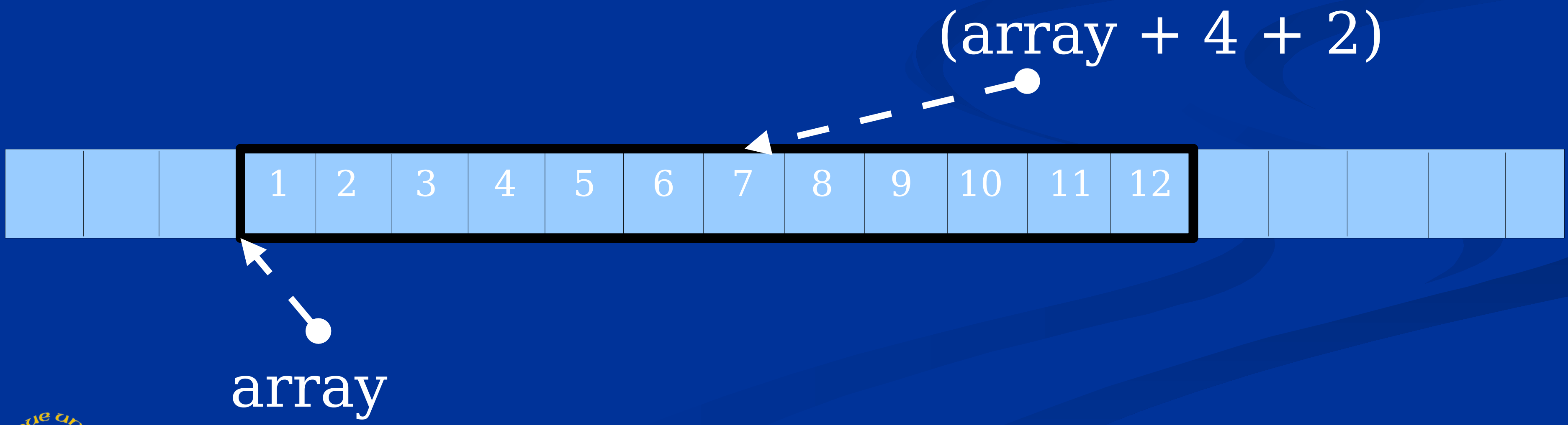
- We can only use dynamic arrays if we know the amount of memory needed before inserting the data into memory
- If we cannot satisfy this condition, we must use linked lists or another dynamic data structure
- Allocate/clear 2D array rows by cols

```
*array = NULL;

array = calloc(sizeof(int), rows * cols);
assert(array != NULL); // or assert(array);
```

Access of dynamic 2D

- How can one access row i, column j?
  `*(array + cols * i + j)`

- For example, cols = 4, i = 1, j = 2:



Dynamic 2D notes

- By dynamically creating a 2D “array” how does the compiler know that it needs to be accessed as a 2D array?
  - It DOESN’T!
  - You must do the pointer arithmetic (and do it correctly)
- Access is truly only valid if:
  0 < i < rows and 0 < j < cols
- What happens outside this range?
  `*(array + rows * i + j)`
- (Remember “shooting yourself in the foot”?)

Are there other ways?

- Using an array of pointers?
- Using a pointer to a pointer?
  - Multiple malloc()s?
  - A single malloc()?
- How much memory should you allocate in each case?

For next lecture

- Start Homework 9
- (re-) read 5.7-5.9 in K&R
    - Beej's 12.6
- Read 1.10, 2.1, 2.2, 2.4, 2.7, 4.4, 4.6, 4.7, A4, A8.1 in K&R :-)
    - Beej's Ch. 6 and 12.11 (incomplete)

Boiler Up!