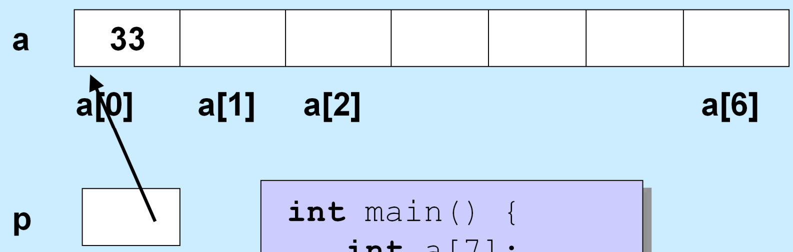
CS 33

Introduction to C
Part 2

Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Some of this lecture is based on material prepared by Pascal Van Hentenryck.

Pointers and Arrays



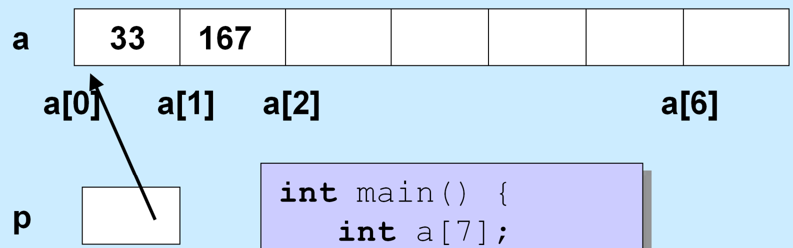

```c
int main() {
   int a[7];
   int *p;
   p = &a[0];
   *p = 33;
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Pointer Arithmetic

Pointers can be incremented/decremented

– what this does to the pointer depends on its type



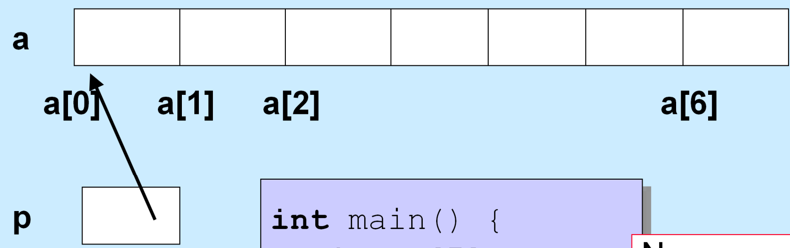

```
int main() {
   int a[7];
   int *p;
   p = &a[0];
   *p = 33;
   *(p+1) = 167;
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Pointer Arithmetic

Pointers can be incremented/decremented

– **what this does to the pointer depends on its type**



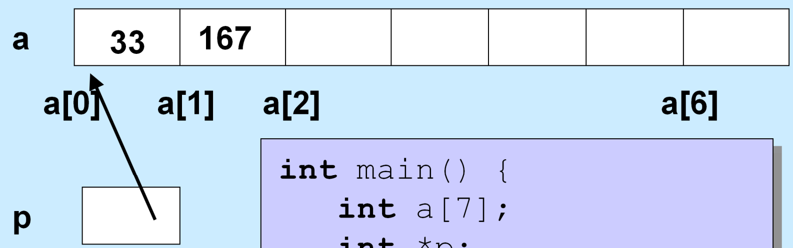

```
int main() {
   int a[7];
   int *p;
   p = &a[0];
}
```

Now `p` and `a` have the same value

Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Pointer Arithmetic

- **Pointers can be incremented/decremented**
  - **what this does to the pointer depends on its type**




```c
int main() {
   int a[7];
   int *p;
   p = a;
   *p = 33;
   p[1] = 167;
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Pointers and Arrays

`p = &a[0];` can also be written as `p = a;`

`a[i];` really is `*(a+i)`

- This is weird and confusing ...
  - `p` is of type `int *`
    - it can be assigned to
      ```
      int *q;
      p = q;
      ```
  - `a` sort of behaves like an `int *`
    - but it can't be assigned to
      ~~a = q;~~

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Pointers and Arrays

- **An array name represents a pointer to the first element of the array**
- **Just like a literal represents its associated value**
  - **in:**

    ```
    x = y + 2;
    ```

    - » **“2” is a *literal* that represents the value 2**
  - **can’t do**

    ```
    2 = x + y;
    ```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Literals and Procedures

```
int proc(int x) {
   x = x + 4;
   return x * 2;
}

int main() {
   result = proc(2);
   printf("%d\n", result);
   return 0;
}
```

initialized with a copy of the argument

Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Arrays and Procedures

initialized with a copy of the argument

```
int proc(int *a, int nelements) {
   int i;
   for (i=0; i<nelements-1; i++)
      a[i+1] += a[i];
   return a[nelements-1];
}

int main() {
   int array[50] = ... ;
   printf("result = %d\n", proc(array, 50));
   return 0;
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Note that the argument to proc is not the entire array, but the pointer to its first element. Thus *a* is initialized by copying into it this pointer.

Equivalently ...

```
int proc(int a[], int nelements) {
   ...
}

int main() {
   ...
}
```

No need for array size, since all that's used is pointer to first element

Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Note that one could include the size of the array ("**int** proc(**int** a[50], **int** nelements)"), but the size would be ignored, since it's not relevant: arrays don't know how big they are. Thus the *nelements* argument is very important.

Arrays and Parameters

```
void func(int arg[]) {
   /* arg points to the caller's array */
   int local[7];      /* seven ints */
   arg++;             /* legal */
   arg = local;       /* legal */
   local++;           /* illegal */
   local = arg;       /* illegal */
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Dereferencing C Pointers

```
int main() {
   int *p; int a = 4;
   p = &a;
   (*p)++;
   printf("%d %u\n", *p, p);
}
```

```
% ./a.out
5 3221224356
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Dereferencing C Pointers

```
int main() {
   int *p; int a = 4;
   p = &a;
   *p++;
   printf("%d %u\n", *p, p);
}
```

```
% ./a.out
3221224360 3221224360
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Operator precedence is hard to remember!

Dereferencing C Pointers

```
int main() {
   int *p; int a = 4;
   p = &a;
   ++*p;
   printf("%d %u\n", *p, p);
}
```

```
% ./a.out
5 3221224356
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

2-D Arrays

- **Suppose `T` is a datatype (such as `int`)**
- **`T n[6]`**
  - **declares `n` to be an array of (six) `T`**
  - **the type of `n` is `T[6]`**
- **Thus `T[6]` is effectively a datatype**
- **Thus we can have an array of `T[6]`**
- **`T m[7][6]`**
  - **`m` is an array of (seven) `T[6]`**
  - **`m[i]` is of type `T[6]`**
  - **`m[i][j]` is of type `T`**

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Note that even though we might think of “int [6]” as being a datatype, to declare “n” to be of that type, we must write “int n[6]” — the identifier we are declaring goes in the middle of the name of the datatype. Similarly, to have an array of seven of this type, we must write “int m[7][6]” — the array indication goes immediately to the right of the name of the identifier. We could have an array of eight of these 2-D arrays; such a 3-D array would be declared “int p[8][7][6]”.

2-D Arrays

```
#define NUM_ROWS 3
#define NUM_COLS 4
…
int main() {
   int row, col;
   int m[NUM_ROWS][NUM_COLS];
   for(row=0; row<NUM_ROWS; row++)
     for(col=0; col<NUM_COLS; col++)
        m[row][col] = row*NUM_COLS+col;
   printMatrix(NUM_ROWS, NUM_COLS, m);
   return 0;
}
```

```
% ./a.out
   0    1    2    3
   4    5    6    7
   8    9    10   11
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

2-D Arrays

It must be told the dimensions

```
void printMatrix(int nr, int nc,
      int m[nr][nc]) {
   int row, col;
   for(row=0; row<nr; row++) {
     for(col=0; col<nc; col++)
        printf("%6d", m[row][col]);
     printf("\n");
   }
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

2-D Arrays

Alternatively …

```
void printMatrix(int nr, int nc,
      int m[][nc]) {
   int row, col;
   for(row=0; row<nr; row++) {
     for(col=0; col<nc; col++)
        printf("%6d", m[row][col]);
     printf("\n");
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

2-D Arrays

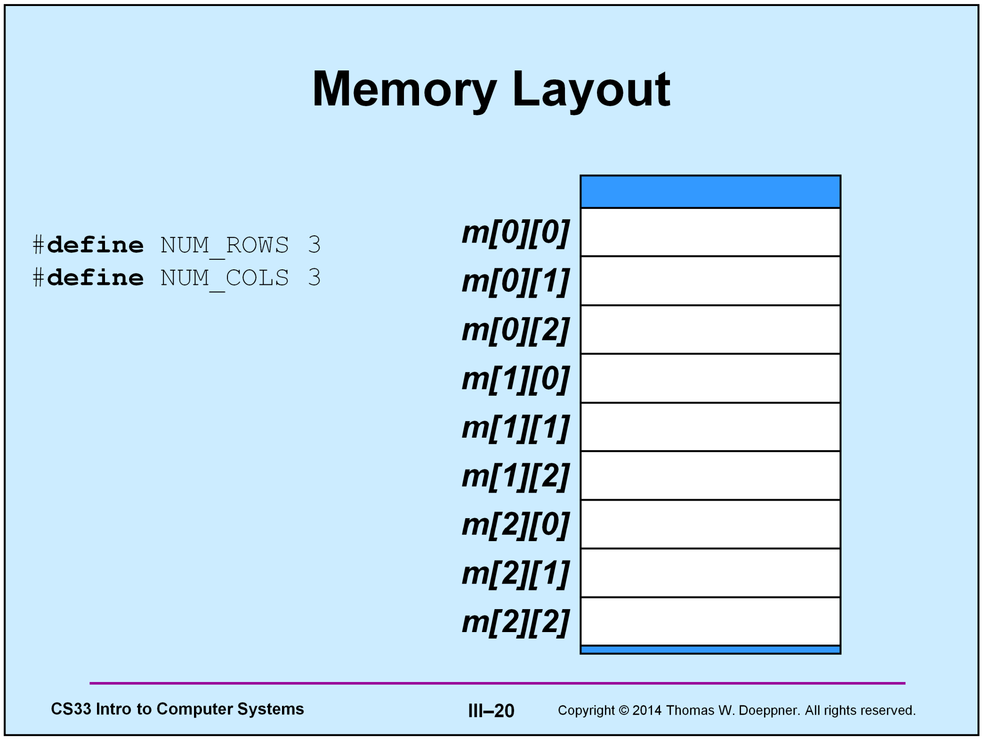
Or …

```
void printMatrix(int nr, int nc,
      int m[nr][nc]) {
   int i;
   for(i=0; i<nr; i++)
      printArray(nc, m[i]);
}
```

```
void printArray(int nc, int a[nc]) {
   int i;
   for(i=0; i<nc; i++)
      printf("%6d", a[i]);
   printf("\n");
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Note that m is an array of arrays (in particular, an array of 1-D arrays).



C arrays are stored in *row-major order*, as shown in the slide. The idea is that the left index references the row, the right index references the column. Thus C arrays are stored row-by-row.

Parameters

```
void func1(int A[], int size);
void func2(int *A, int size);
  /* both work fine */

void func3(int A[][], int r, int c);
void func4(int **A, int r, int c);
  /* no good: compiler doesn't know
     the size of A's rows */
void func5(int A[][3], int r);
void func6(int r, int c, int A[][c]);
  /* both good: row sizes are known */
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

A Bit More Syntax …

- **Constants**

```
const double pi =
  3.14159265358979323 8;
```

```
area = pi*r*r;     /* legal */
pi = 3.0;          /* illegal */
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

More Syntax …

```
const int six = 6;
int nonconstant;
const int *ptr_to_constant;
int *const constant_ptr = &nonconstant;
const int *const constant_ptr_to_constant = &six;

ptr_to_constant = &six;
   // ok
*ptr_to_constant = 7;
   // not ok
*constant_ptr = 7;
   // ok
constant_ptr = &six;
   // not ok
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Note that constant_ptr_to_constant's value may not be changed, and the value of what it points to may not be changed.

And Still More …

- **Array initialization**

```
int FirstSixPrimes[6] = {2, 3, 5, 7, 11, 13};
int SomeMorePrimes[] = {17, 19, 23, 29};
int MoreWithRoomForGrowth[10] = {31, 37};
int MagicSquare[][] = {{2, 7, 6},
                       {9, 5, 1},
                       {4, 3, 8}};
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Global Variables

The scope is global;
m can be used
by all functions

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   int row, col;
   for(row=0; row<NUM_ROWS; row++)
     for(col=0; col<NUM_COLS; col++)
        m[row][col] = row*NUM_COLS+col;
   return 0;
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Global Variables

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   int row, col;
   printf("%u\n", m);
   printf("%u\n", &row);
   return 0;
}
```

```
% ./a.out
8384
3221224352
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Note that the reference to “m” gives the address of the array in memory.

Global Variables are Initialized!

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   printf("%d\n", m[0][0]);
   return 0;
}
```

```
% ./a.out
0
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Scope

```
int a;   // global variable

int main() {
   int a;   // local variable
   a = 0;
   proc();
   printf("a = %d\n", a); // what's printed?
   return 0;
}

int proc() {
   a = 1;
   return a;
}
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Hint: the answer is not 1.

Scope (continued)

```
int a;   // global variable

int main() {
   a = 0;
   proc(1);
   return 0;
}

int proc(int a) {
   printf("a = %d\n", a); // what’s printed?
   return a;
}
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Hint: the answer is not 0.

Scope (still continued)

```
int a;   // global variable

int main() {
   a = 0;
   proc(1);
   return 0;
}

int proc(int a) {
   int a;
   printf("a = %d\n", a); // what's printed?
   return a;
}
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Syntax error …

Scope (more ...)

```
int a;   // global variable

int main() {
   {
      // the brackets define a new scope
      int a;
      a = 6;
   }
   printf("a = %d\n", a); // what's printed?
   return 0;
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

0

Lifetime

```
int count;

int main() {
   func();
   ...
   func(); // what’s printed by func?
   return 0;
}

int func() {
   int a;
   if (count == 0) a = 1;
   count = count + 1;
   printf("a = %d\n", a);
   return 0;
}
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

undefined.

Lifetime (continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}
int a;
int func(int x) {
   if (x == 1) {
      a = 1;
      func(2);
      printf("a = %d\n", a);
   } else
      a = 2;
   return 0;
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Lifetime (still continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}

int func(int x) {
   int a;
   if (x == 1) {
      a = 1;
      func(2);
      printf("a = %d\n", a);
   } else
      a = 2;
   return 0;
}
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Lifetime (more ...)

```
int main() {
   int *a;
   a = func();
   printf("*a = %d\n", *a); // what’s printed?
   return 0;
}

int *func() {
   int x;
   x = 1;
   return &x;
}
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

undefined.

Lifetime (and still more ...)

```
int main() {
   int *a;
   a = func(1);
   printf("*a = %d\n", *a); // what's printed?
   return 0;
}

int *func(int x) {
   return &x;
}
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

undefined.

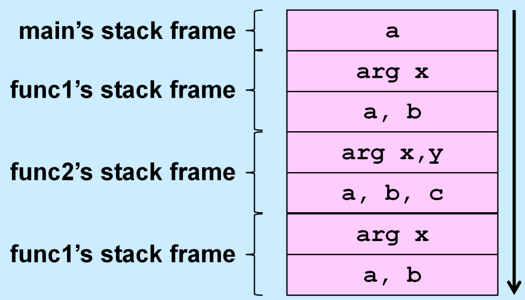
Rules

- Global variables exist for the duration of program's lifetime
- Local variables and arguments exist for the duration of the execution of the procedure
  - from call to return
  - each execution of a procedure results in a new instance of its arguments and local variables

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Implementation: Stacks

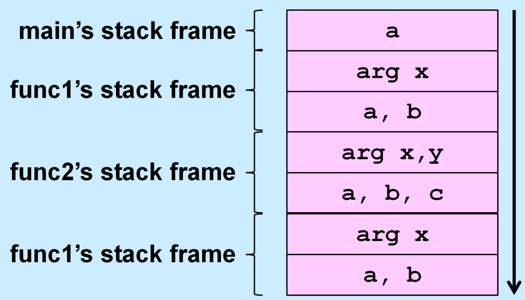
```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Implementation: Stacks

```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Copyright © 2014 Thomas W. Doeppner. All rights reserved.

scanf: Reading Data

```
int main() {
   int i, j;
   scanf("%d %d", &i, &j);
}
```

Two parts

- formatting instructions
- arguments: must be addresses
  - why?

Copyright © 2014 Thomas W. Doeppner. All rights reserved.

#define (again)

```
#define CtoF(cent) (9.0*cent)/5.0 + 32.0
```

Simple textual substitution:

```
float tempc = 20.0;
float tempf = CtoF(tempc);
// same as tempf = (9.0*tempc)/5.0 + 32.0;
```


Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Careful ...

```
#define CtoF(cent) (9.0*cent)/5.0 + 32.0
```

```
float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*tempc+10)/5.0 + 32.0;
```

```
#define CtoF(cent) (9.0*(cent))/5.0 + 32.0
```

```
float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*(tempc+10))/5.0 + 32.0;
```

 Copyright © 2014 Thomas W. Doeppner. All rights reserved.

Be careful with how arguments are used! Note the use of parentheses in the second version.