Pointers, Arrays and Strings in C

Systems Programming

Outline

- **What exactly are variables?**
- **Pointers**
  - Pointer variables and NULL
  - Operators
  - Pointer arithmetic
  - Pointers to structures (+ pseudo-polymorphism)
  - Pointers to arrays
  - void pointers
  - Dynamic allocation of memory
- **Arrays**
  - Array names in C
  - Array indexing and pointer arithmetic
  - Multidimensional arrays
- **Strings**
  - Global and local strings

What exactly are variables? (1/4)

- A **variable** in a program is **something with a name**, the **value of which can vary**

- The way the compiler and linker handles this is that it assigns a specific **block of memory** to hold the value of that variable

- When we declare a variable we inform the compiler of:
  - » the **name** of the variable, and
  - » the **type** of the variable

What exactly are variables? (2/4)

- For example, we declare an integer variable with the name **k** by writing:

```
int k;
```

  - The compiler sets aside **4 bytes of memory** to hold the value (IA-32)
  - It also sets up a *symbol table*, and adds the symbol **k** and the relative address in memory where those 4 bytes were set aside
  - Thus, later if we write: `k = 2133;` we expect that, at run time when this statement is executed, the value 2133 will be placed in that memory location reserved for the storage of the value of **k**
  - In C we refer to a variable such as the integer **k** as an "**object**"

What exactly are variables? (3/4)

- In our previous example, in a sense there are two "values" associated with the object **k**
  - One is the **value** of the integer stored there (2133 in the above example)
  - The other the "value" of the memory location, i.e., the **address** of **k**
- Some texts refer to these two values respectively as
  - ***rvalue*** (right value, pronounced "are value") **= value** and
  - ***lvalue*** (left value, pronounced "el value") **= address**

What exactly are variables? (4/4)

- In some languages, the **lvalue** is the value permitted on the left side of the assignment operator '='
  - » i.e. the address where the result of evaluation of the right side ends up
- The **rvalue** is that which is on the right side of the assignment statement, the **2133** above
  - rvalues cannot be used on the left side of the assignment statement; thus: **2 = k**; is illegal
- Actually, the above definition of "lvalue" is somewhat modified for C, but we will come to this again later

Pointer variables

- A **pointer variable** is a variable to hold an **lvalue** (an **address**)
  - » The size required to hold such a value depends on the system
  - » The actual size required is not too important, as long as we have a way of informing the compiler that what we want to store is an address

- Pointer variables in C:
  - » We define a pointer variable by preceding its name with an asterisk
  - » In C we also give our pointer a **type** which refers to the type of data stored at the address we will be storing in our pointer.
  - » Example:

```
int *ptr;
```

  - » Such a pointer is said to "point to" an integer

The "null" pointer and **`NULL`**

- If the definition of a pointer variable is made outside of any function (=global/static), ANSI-compliant compilers will initialize it to a value guaranteed to not point to any C object or function
  - » A pointer initialized in this manner is called a **"null" pointer**.
- **Be careful!** A null pointer may or may not evaluate to zero
  - » This depends on the specific system on which the code is developed.
    - Normally (=everywhere known) this is true; but technically (=ANSI C) 0 converted to a pointer is a "null pointer constant" pointing to no object; but it need not be all 0 bits…
    - Therefore, and for lots of other good reasons, C++ (in C++11) introduced **`nullptr`**, a new constant of a new type (**`nullptr_t`**)
  - » For source compatibility, a macro is used to represent a null pointer
  - » That macro goes under the name **`NULL`** and should **always** be used!
- To guarantee that a pointer has become a null pointer we set its value using the **`NULL`** macro: **`ptr = NULL;`**
- Similarly, we can test for a null pointer:

`if (ptr == NULL) ...` **or even** **`if (!ptr) ...`**

Pointer-related operators

- To store in **`ptr`** the **address** of our integer variable **`k`**, we use the unary **&** (address-of) operator and write:

  ```
  ptr = &k;
  ```

  - The **&** operator **retrieves the lvalue** (address) of **`k`**
  - The assignment operator '=' copies that to the contents of ptr
  - Now, ptr is said to "point to" **`k`**

- The **dereferencing operator** is the asterisk; and it is used as follows:

  ```
  *ptr = 7;
  ```

  - This statement will copy 7 to the address pointed by **`ptr`**. Thus if **`ptr`** "points to" **`k`**, the above statement will set the value of **`k`** to 7.

- When we use the * this way, we are referring to the **value of that which ptr points to**, not the value of the pointer **itself!**

Putting it all together – pointer_values.c

```c
#include <stdio.h>

int j, k;
int *ptr;

int main(void) {
    j = 1;
    k = 2;
    ptr = &k;
    printf("\n");
    printf("j has the value %d and is stored at %p\n", j, &j);
    printf("k has the value %d and is stored at %p\n", k, &k);
    printf("ptr has the value %p and is stored at %p\n", ptr, &ptr);
    printf("The value of the integer pointed to by ptr is %d\n", *ptr);
}
```

```
j has the value 1 and is stored at 0x601048
k has the value 2 and is stored at 0x601058
ptr has the value 0x601058 and is stored at 0x601050
The value of the integer pointed to by ptr is 2
```

Pointer types

- Why do we need to identify the ***type*** of variable that a pointer points to?

- One reason for this is, that the compiler knows how many bytes to copy into or out of that memory location in the future
  - » For example, if **`ptr`** is an integer pointer, **`*ptr = 2;`** would result in 4 bytes being copied into the memory location contained in **`ptr`**
    - » Assuming that an integer takes up 4 bytes

- But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code

Basic pointer arithmetic (1/2)

- Consider a block in memory consisting of ten integers in a row
  - That is, 40 bytes of memory are set aside to hold 10 integers

- Now, let's say we point our integer pointer **`ptr`** at the first of these integers

- Furthermore lets say that the first integer is at memory location 4000 (decimal). What happens when we write:

```
ptr + 1;
```

  - Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer (its current value, 4000, is the address of an integer), it adds **4** to **`ptr`** instead of 1, so the pointer "points to" the **next integer**, at memory location 4004.

Basic pointer arithmetic (2/2)

- This works similarly for other data types such as floats, doubles, or even user defined data types such as structures
- This is obviously not the same kind of "addition" that we normally think of. In C it is referred to as addition using **"pointer arithmetic"**, a term which we will come back to later

Pointer and arrays

- Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting **relationship between arrays and pointers**

- Consider the following:

```
int my_array[] = {1,23,17,4,-5,100};
```

This is an array containing 6 integers. We refer to each of these integers by means of a subscript to **my_array**, i.e. using **`my_array[0]`** through **`my_array[5]`**. But, we could alternatively access them via a pointer:

```
int *ptr;
ptr = &my_array[0]; /* point our pointer at the    */
                    /* first integer in our array */
```

and then we could print out our array either using the array notation or by dereferencing our pointer

What exactly are array names? (1/2)

- In C, the standard states that wherever we might use **`&var_name[0]`** we can replace that with **`var_name`**, thus in our code where we wrote:

  ```
  ptr = &my_array[0];
  ```

  we can write:

  ```
  ptr = my_array;
  ```

- One way of looking at it is that the **name of an array is a (constant) pointer**

- Another (maybe more intuitive) is that **the name of the array is the address of the first element in the array**

What exactly are array names? (2/2)

- Note that, while we can write

  ```
  ptr = my_array;
  ```

  we **cannot** write

  ```
  my_array = ptr;
  ```

- The reason is that while `ptr` is a variable, `my_array` is a constant.
  - » The location at which the first element of `my_array` will be stored cannot be changed once `my_array[]` has been defined.

lvalues in C

- K&R-2 (sort of the "bible" of C) states:

  "An **object** is a named region of storage;
  an **lvalue** is an expression referring to an object".

- Now, although **`my_array`** is a named region of storage, in the following assignment statement, it is **not** an lvalue!

```
my_array = ptr; /* error! */
```

- To resolve this problem, some refer to **`my_array`** as an "unmodifiable lvalue".

Example – indexing.c

```c
#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void) {

    int i;
    ptr = &my_array[0]; /* point pointer to the first element of the array */
    printf("\n\n");

    for (i = 0; i < 6; i++) {
        printf("my_array[%d] = %d ", i, my_array[i]);     /*<-- A */
        printf("\t ptr + %d = %d\n", i, *(ptr + i));     /*<-- B */
    }
}
```

```
my_array[0] = 1          ptr + 0 = 1
my_array[1] = 23         ptr + 1 = 23
my_array[2] = 17         ptr + 2 = 17
my_array[3] = 4          ptr + 3 = 4
my_array[4] = -5         ptr + 4 = -5
my_array[5] = 100        ptr + 5 = 100
```

Array indexing and pointer arithmetic (1/2)

- Both the character pointer and the name of the array give us an **address of the first element** of the array
  - The **value** behind this address is the same whether we dereference a character pointer or index an array name
  - This implies that somehow **my_array[i]** is the same as ***(ptr + i)**
- Wherever one writes **a[i]** it can be replaced with ***(a + i)** without any problems. Thus we see that **pointer arithmetic is the same thing as array indexing** (or the reverse!).
  - **Note**: this does NOT mean that pointers and arrays are the same thing, they are not
  - We are only saying that to identify a given element of an array we have the choice of two syntaxes which yield identical results
- In the expression **(a + i)**, part of it is a simple addition using the **+** operator. The rules of C state that such an expression is commutative. That is **(a + i)** is identical to **(i + a)**.
  - Thus we could write ***(i + a)** just as easily as ***(a + i)**

You could do, but shouldn’t – reversed_indexing.c

- Look at this program:

```c
#include <stdio.h>

char str[80] = "A string to be used for demonstration purposes";

int main(void) {
    printf("The third character of the string is: '%c'\n", str[3]);
    printf("The third character of the string is: '%c'\n", *(str + 3));
    printf("The third character of the string is: '%c'\n", *(3 + str));
    printf("The third character of the string is: '%c'\n", 3[str]); /* <-- !!! */
}
```

```
The third character of the string is: 't'
The third character of the string is: 't'
The third character of the string is: 't'
The third character of the string is: 't'
```

» How can this work ?!?!

» **a[i]** is equivalent to ***(a + i)** which is equivalent to ***(i + a)** which should be (and is!) equivalent to **i[a]**!

» Compare to Assembler: **movq $3,%rdi** **movb str(,%rdi,1), %al**
movq $str,%rdi **movb 3(,%rdi,1), %al**

Dereferencing and operator precedence (1/2)

- In the indexing.c example note lines A and B and that the program prints out the same values in either case
  - Also observe how we dereferenced our pointer in line B: we first added **`i`** to it and then dereferenced the new pointer

- What do you think happens if you change line B to read:

```
printf("\t ptr + %d = %d\n", i, *ptr++);
```

- And what if you change it to:

```
printf("\t ptr + %d = %d\n", i, *(++ptr));
```

- And where will **`ptr`** point to after the loop in both cases?

Dereferencing and operator precedence (2/2)

```
printf("\t ptr + %d = %d\n", i, *ptr++);
```

```
my_array[0] = 1         ptr + 0 = 1
my_array[1] = 23        ptr + 1 = 23
my_array[2] = 17        ptr + 2 = 17
my_array[3] = 4         ptr + 3 = 4
my_array[4] = -5        ptr + 4 = -5
my_array[5] = 100       ptr + 5 = 100
```

```
printf("\t ptr + %d = %d\n", i, *(++ptr));
```

```
my_array[0] = 1         ptr + 0 = 23
my_array[1] = 23        ptr + 1 = 17
my_array[2] = 17        ptr + 2 = 4
my_array[3] = 4         ptr + 3 = -5
my_array[4] = -5        ptr + 4 = 100
my_array[5] = 100       ptr + 5 = 0
```

Final position of `ptr`: **after** the last array element (dangerous!)

Strings in C (1/2)

- In C, **strings are arrays of characters**
  - Specifically, in C a string is an array of characters **terminated with a binary zero character** (written as **'\0'**)
- Consider, this example:

```
char my_string[40];
my_string[0] = 'T';
my_string[1] = 'e';
my_string[2] = 'd':
my_string[3] = '\0';
```

- While one would never build a string like this, the end result is a string in that it is an array of characters **terminated with a nul character**
- Be aware that "nul" is **not** the same as "NULL“!
  - » **nul**
    - » Refers to a zero as defined by **'\0'**, and occupies one byte of memory
    - » nul may not be **`#defined`** at all
  - » **NULL**
    - » Name of the macro used to initialize null pointers
    - » **`#defined`** in a header file in your C compiler

Strings in C (2/2)

- Since writing the above code would be very time consuming, C permits two **alternate ways** of achieving the same thing
  - » First, one might write:

    ```
    char my_string[40] = {'T', 'e', 'd', '\0',};
    ```

  - » But this also takes more typing than is convenient. So, C permits:

    ```
    char my_string[40] = "Ted";
    ```

  - » When the double quotes are used, the nul character ('`\0`') is automatically appended to the end of the string

- In all of the above cases, the same thing happens:
  - » The compiler sets aside an contiguous block of memory 40 bytes long to hold characters and
  - » Initializes it such that the first 4 characters are `Ted\0`.

Global strings (1/2)

- “Global” in this context means “outside any function, including **`main()`**”

- A declaration like

```
char my_string[40] = "Ted";
```

  would allocate space for a 40 byte array and put the string in the first 4 bytes (three for the characters in the quotes and a 4th to handle the terminating **'\0'**)

- Actually, if we just wanted to store the name "Ted" we could write:

```
char my_name[] = "Ted";
```

  and the compiler would count the characters, leave room for the nul character and store these four characters in memory, the location of which would be returned by the array name, in this case **`my_name`**

Global strings (2/2)

- An alternative way to achieve the same effect would be:

```
char *my_name = "Ted";
```

- But there are **important differences** between the two approaches!
  - Using the array notation, 4 bytes of storage in the static memory block are taken up, one for each character + one for the terminating nul char
  - But in the pointer notation, the same 4 bytes are required, **plus k bytes** to store the pointer variable **`my_name`** (where k depends on the system)
  - In the array notation, **`my_name`** is short for **`&myname[0]`** which is the address of the first element of the array. Since the location of the array is fixed during run time, this is a constant (not a variable)
  - In the pointer notation **`my_name`** is a variable (and so requires separate space; but also can be modified, i.e. point to a different string/address)

- Neither of the two approaches is a "better" method; which one you should use depends on what you are going to do within the rest of the program

Local strings

- Now consider similar declarations made within a function:

```
void my_function_A(char **ptr) {
    char a[] = "ABCDE";
    ...
}

void my_function_B(char **ptr) {
    char *cp = "FGHIJ";
    ...
}
```

  - » In the case of **my_function_A**, the content, or value(s), of the array **`a[]`** is considered to be the data. The array is said to be initialized to the values **`ABCDE`**.
  - » In the case of **my_function_B**, the value of the pointer **`cp`** is considered to be the data. The pointer has been initialized to point to the string **`FGHIJ`**.
  - » In both **my_function_A** and **my_function_B** the definitions are local variables, so the string **`ABCDE`** is stored on the stack, as is the value of the pointer **`cp`**. **But the string `FGHIJ` will be stored in the data section!**

Local strings

Similar program as before:

```c
void my_function_A(char **ptr) {
    char a[] = "ABCDE";
    *ptr = a;
}

void my_function_B(char **ptr) {
    char *cp = "FGHIJ";
    *ptr = cp;
}

int main(void) {
    char *test;
    my_function_A(&test);
    printf("Function A: %s\n", test);
    my_function_B(&test);
    printf("Function B: %s\n", test);
    return 0;
}
```

```
Function A: ����z��?��"���?��`��?���ui���|�k�?��`�
Function B: FGHIJ
```

Example with strings – strcpy.c

```c
#include <stdio.h>

char strA[80] = "A string to be used for demonstration purposes";
char strB[80];

int main(void) {
    char *pA;    /* a pointer to type character */
    char *pB;    /* another pointer to type character */
    puts(strA); /* show string A */
    pA = strA;  /* point pA at string A */
    puts(pA);   /* show what pA is pointing to */
    pB = strB;  /* point pB at string B */
    putchar('\n'); /* move down one line on the screen */
    while (*pA != '\0') { /* line A (see text) */
        *pB++ = *pA++; /* line B (see text) */
    }
    *pB = '\0'; /* line C (see text) */
    puts(strB); /* show strB on screen */
}
```

```
A string to be used for demonstration purposes
A string to be used for demonstration purposes

A string to be used for demonstration purposes
```

A custom implementation of **`strcpy()`**

With what we have learned, we can create our own replacement for the **`strcpy()`** function that comes with the standard C library

A simplified version might look like this:

```
char *my_strcpy(char *destination, char *source) {
    char *p = destination;
    while (*source != '\0')  {
        *p++ = *source++;
    }
    *p = '\0';
    return destination;
}
```

and, using it:

```
char strA[80] = "A string to be used for demonstration purposes";
char strB[80];

char *my_strcpy(char *destination, char *source);

int main(void) {
    my_strcpy(strB, strA);
    puts(strB);
}
```

A variation of `my_strcpy()` – my_strcpy_array.c

- Consider this variation of **`my_strcpy()`**:

```
char *my_strcpy(char dest[], char source[]) {
    int i = 0;
    while (source[i] != '\0') {
        dest[i] = source[i];
        i++;
    }
    dest[i] = '\0';
    return dest;
}
```

  - » Recall that strings are arrays of characters. Here we have chosen to use **array notation** instead of pointer notation to do the actual copying.
  - » The results are the same, i.e. the string gets copied using this notation just as accurately as it did before

Pointers to structures

- Similarly to what we have seen before, we **declare a pointer** to a structure like this:

```
struct tag *st_ptr;
```

  and we **point it to an existing structure** with:

```
st_ptr = &my_struct;
```

- The syntax for **accessing members of a structure** through a pointer:

```
(*ptr_var_name).member_name
```

  - » For example, to use the pointer in the following example to set the age of the employee, we would write:

```
(*st_ptr).age = 63;
```

- However, this is a fairly often used expression and the designers of C have created an **alternate syntax** with the same meaning which is:

```
st_ptr->age = 63;
```

- `'->'` is the **pointer operator**.

Example structure – person_struct.c

- A simple example with structures:

```c
#include <stdio.h>
#include <string.h>

struct person {
    char lname[20]; /* last name */
    char fname[20]; /* first name */
    int age; /* age */
    float rate; /* e.g. 12.75 per hour */
};

struct person my_struct; /* declare the structure my_struct */

int main(void) {
    strcpy(my_struct.lname, "Jensen");
    strcpy(my_struct.fname, "Adam");
    printf("\n%s ", my_struct.fname);
    printf("%s\n", my_struct.lname);
}
```

- Note that, here, to access the member of the structure, we use the dot operator ('.')

Adding pointers… – person_pointer.c

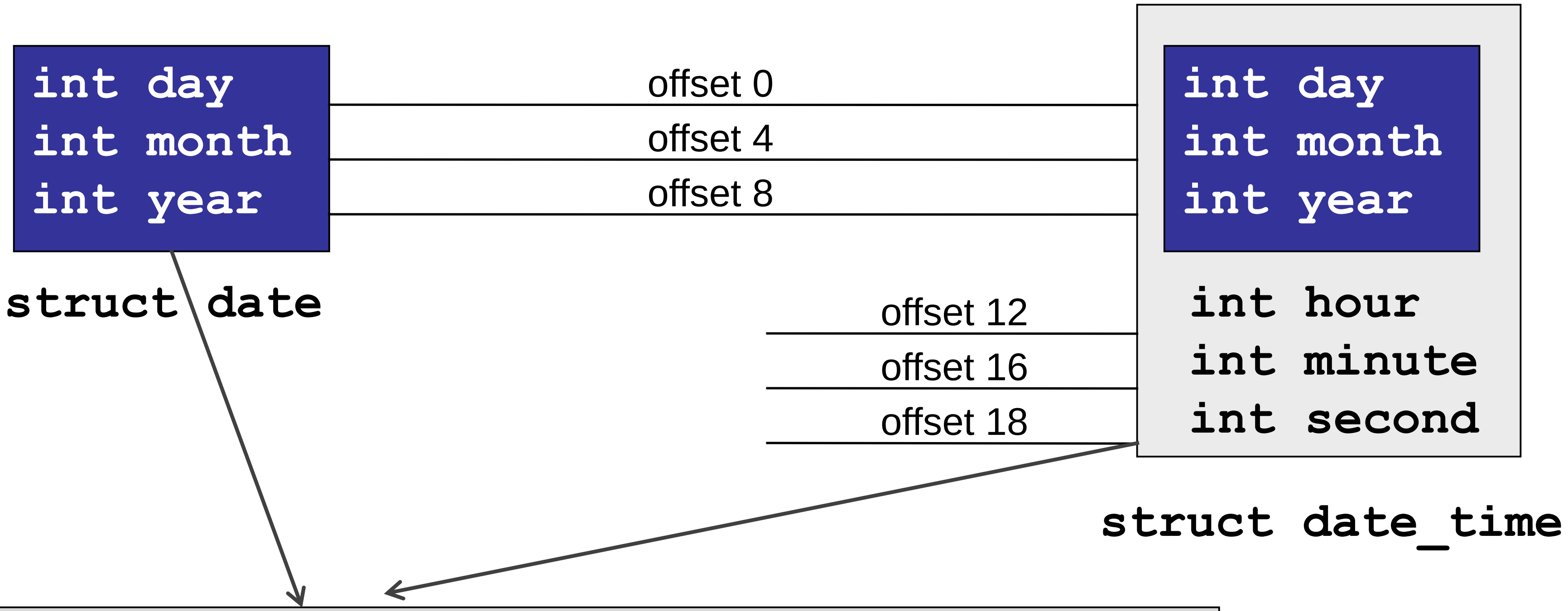
```c
#include <stdio.h>
#include <string.h>

struct tag{ /* the structure type */
    char lname[20]; /* last name */
    char fname[20]; /* first name */
    int age; /* age */
    float rate; /* e.g. 12.75 per hour */
};
struct tag my_struct; /* define the structure */
void show_name(struct tag *p); /* function prototype */

int main(void) {
    struct tag *st_ptr; /* a pointer to a structure */
    st_ptr = &my_struct; /* point the pointer to my_struct */
    strcpy(my_struct.lname, "Jensen");
    strcpy(my_struct.fname, "Adam");
    printf("\n%s ", my_struct.fname);
    printf("%s\n", my_struct.lname);
    my_struct.age = 27;
    show_name(st_ptr); /* pass the pointer */
}

void show_name(struct tag *p) {
    printf("\n%s ", p->fname); /* p points to a structure */
    printf("%s ", p->lname);
    printf("%d\n", p->age);
}
```

Structures and pseudo-polymorphism (1/4)



```
print(struct date d){
   printf("%d-%d-%d", d.day, d.month, d.year);
}
```

- We can take advantage of the fact that member access in structures is based on the member's offset, to achieve a kind of “polymorphism”
- I.e., we can create structures that can be “cast to”, and used as, some “base” structure

Structures and pseudo-polymorphism (2/4)

- Reminder:
  - Whenever we access one of their members, what we are actually accessing is a typed variable, whose memory location is defined as an *offset*, relatively to the address of the structure variable itself

```c
#include <stddef.h>

struct date {
    int day;
    int month;
    int year;
};

main () {
    printf("offset of date.day: %d\n", offsetof(struct date, day));
    printf("offset of date.month: %d\n", offsetof(struct date, month));
    printf("offset of date.year: %d\n", offsetof(struct date, year));
}
```

```
offset of date.day: 0
offset of date.month: 4
offset of date.year: 8
```

Structures and pseudo-polymorphism (3/4)

Example 1

```
offset of date.day: 0
offset of date.month: 4
offset of date.year: 8
offset of date_time.date.day: 0
offset of date_time.date.month: 4
offset of date_time.date.year: 8
```

```
#include <stddef.h>

struct date {
    int day;
    int month;
    int year;
};

struct date_time {
    struct date date;
    int hour;
    int minute;
    int second;
};

main () {
    printf("offset of date.day: %d\n", offsetof(struct date, day));
    printf("offset of date.month: %d\n", offsetof(struct date, month));
    printf("offset of date.year: %d\n", offsetof(struct date, year));

    printf("offset of date_time.date.day: %d\n", offsetof(struct date_time, date.day));
    printf("offset of date_time.date.month: %d\n", offsetof(struct date_time, date.month));
    printf("offset of date_time.date.year: %d\n", offsetof(struct date_time, date.year));
}
```

Structures and pseudo-polymorphism (4/4)

Example 2

```c
#include <stddef.h>

struct date {
    int day;
    int month;
    int year;
};

struct date_time {
    struct date date;
    int hour;
    int minute;
    int second;
};

void set_date(struct date* dptr) {
    dptr->day = 1;
    dptr->month = 2;
    dptr->year = 2004;
}

main () {
    struct date_time date_time_instance;

    set_date((struct date *)&date_time_instance);
    printf("day: %lu\n", date_time_instance.date.day);
    printf("month: %lu\n", date_time_instance.date.month);
    printf("year: %lu\n", date_time_instance.date.year);
}
```

```
day: 1
month: 2
year: 2004
```

void pointers (1/2)

- Things we have seen so far
  - Pointers can point to data objects of **various types**
  - **Operations on pointers** (e.g., arithmetic, dereferencing) respect the size of its type
  - On different systems these sizes can vary, just like they do for types
- Like with integers where you can run into trouble attempting to assign a long integer to a variable of type short integer, you can run into trouble attempting to assign the values of pointers of various types to pointer variables of other types
- void pointers can be cast to or from any other type of pointer
  - Attention 1: Typecast takes place implicitly!
  - Attention 2: A typecast is necessary in C++, as this automatic conversion was removed

void pointers (2/2)

- To minimize this problem, C provides for pointers of **type void**
- We can declare such a pointer by writing:

```
void *vptr;
```

- A void pointer is sort of a **generic pointer**
  - While C will not permit the comparison of a pointer to type integer with a pointer to type character, either of these can be compared to a void pointer
- Of course, as with other variables, **casts** can be used to convert from one type of pointer to another under the proper circumstances

void *

Pointer to function

```
/* function returning integer */
int func();

/* function returning pointer to integer */
int *func();

/* pointer to function returning integer */
int (*func)();


/* pointer to function returning pointer to integer */
int *(*func)();
```

- Advantages?
  - More flexibility
  - Dynamic programming
  - Simulation of OO!
    - First C++ compilers were “precompilers” generating a C program

Pointer to function – Example (1/2)

pointer_to_function.c

```c
#include <stdio.h>
#include <string.h>

char to_upper(char c);
char to_lower(char c);
void process(char *string, char (* processor_function)(char));

int main(void) {
    char example[12] = "hello world";
    process(example, to_upper);    /* pass function pointer */
    printf("to upper: %s\n", example);
    process(example, to_lower);    /* and again */
    printf("to lower: %s\n", example);
    return 0;
}


...
```

Pointer to function – Example (2/2)

```
...

void process(char *string, char (* processor_function)(char)) {
    int i, size = strlen(string);
    for (i=0; i<size; i++) {
        string[i] = (*processor_function)(string[i]);
    }
}

char to_upper(char c) {
    if (c >= 'a' && c <= 'z')
        c += 'A' - 'a';
    return c;
}

char to_lower(char c) {
    if (c >= 'A' && c <= 'Z')
        c -= 'A' - 'a';
    return c;
}
```

Multidimensional arrays (1/14)

- Consider the following declaration:

  `char multi[5][10];`

- Let's see what it means by breaking it down:

  `char multi[5][10];`

- Let's pretend that the emphasised part is the "name" of an array. Then prepending **`char`** and appending **`[10]`**, we have an array of 10 characters.
- But, the name **`multi[5]`** is itself an array indicating that there are 5 elements each being an array of 10 characters

- Hence we have an array of 5 arrays of 10 characters each

Multidimensional arrays (2/14)

- Assume we have filled this two dimensional array with data of some kind. In memory, it might look as if it had been formed by initializing 5 separate arrays using something like:

```
multi[0] = {'0','1','2','3','4','5','6','7','8','9'};
multi[1] = {'a','b','c','d','e','f','g','h','i','j'};
multi[2] = {'A','B','C','D','E','F','G','H','I','J'};
multi[3] = {'9','8','7','6','5','4','3','2','1','0'};
multi[4] = {'J','I','H','G','F','E','D','C','B','A'};
```

However, this is **not possible**, because initializations of arrays need to happen in the same line as their definition. To initialize a **multi-dimensional array**, use something like:

```
multi = { {'0','1','2','3','4','5','6','7','8','9'}, { ... }, { ... },
  { ... }, {'J','I','H','G','F','E','D','C','B','A'} };
```

- At the same time, individual elements would be addressable using syntax such as:

```
multi[0][3] = '3';
multi[1][7] = 'h';
multi[4][0] = 'J';
```

Multidimensional arrays (3/14)

- Since arrays are contiguous in memory, our actual memory block for the above should look like this:

```
0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA
^
|_____ starting at the address &multi[0][0]
```

- Note that we did **not** write `multi[0] = "0123456789"`. Had we done so a terminating `'\0'` would have been implied, since whenever double quotes are used, a `'\0'` character is appended to the characters contained within those quotes. Had that been the case we would have had to set aside room for 11 characters per row instead of 10.

- What the above example really illustrates is, how memory is laid out for 2-dimensional arrays. That is, this is a 2-dimensional array of characters, **not** an “array of strings”.

Multidimensional arrays (4/14)

- Given the earlier code, the compiler knows how many columns are present in the array so it can interpret **`multi+1`** as the address of the 'a' in the 2nd row. That is, it adds 10, the number of columns, to get this location.
  - If we were dealing with integers and an array with the same dimension, the compiler would add **`10*sizeof(int)`**
- Thus, the address of the **9** in the 4th row above would be **`&multi[3][0]`** or **`*(multi+3)`** in pointer notation. To get to the content of the 2nd element in the 4th row we add 1 to this address and dereference the result as in **`*(*(multi+3)+1)`**
- In general, **`*(*(multi+row)+col)`** and **`multi[row][col]`** yield the same results
- The following example illustrates this using integer arrays instead of character arrays

Multidimensional arrays (5/14) – multidim_1.c

```c
#include <stdio.h>
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];

int main(void) {
    int row, col;
    for (row = 0; row < ROWS; row++) {
        for (col = 0; col < COLS; col++) {
            multi[row][col] = row*col;
        }
    }
    for (row = 0; row < ROWS; row++) {
        for (col = 0; col < COLS; col++) {
            printf("\n[%d][%d]  ", row, col);
            printf("%2d ",multi[row][col]);
            printf("%2d ",*(*(multi + row) + col));
        }
    }
    return 0;
}
```

Multidimensional arrays (6/14)

- Because of the double de-referencing required in the pointer version, the name of a **2-dimensional array** is often said to be equivalent to a **pointer to a pointer**

- With a **three-dimensional array** we would be dealing with an array of arrays of arrays and some might say its name would be equivalent to a **pointer to a pointer to a pointer**

- **However**, here we have initially set aside the block of memory for the array by defining it using array notation. Hence, we are dealing with a constant, not a variable. That is we are talking about a **fixed address not a variable pointer**.

- The dereferencing expression used above permits us to access any element in the array of arrays without the need of changing the value of that address (the address of **`multi[0][0]`** as given by the symbol **`multi`**)

Multidimensional arrays (7/14)

- In the previous example we saw that given this declaration

```
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];
```

- we can access individual elements of the array **multi** using either:

  `multi[row][col]`

  or

  `*(*(multi + row) + col)`

- To understand more fully what is going on, let us examine the second expression more closely

Multidimensional arrays (8/14)

- To start with, we will replace **`*(multi + row)`** with **X** making the second expression:

  `*(X + col)`

- Now, from this we see that **X** is like a pointer since the expression is de-referenced and we know that **`col`** is an integer. Here the arithmetic being used is of the special "pointer arithmetic" kind.
  - » That means that, since we are talking about an integer array, the address pointed by (i.e. value of) **`X + col + 1`** must be greater than the address **`X + col`** by an amount equal to **`sizeof(int)`**.

- Since we know the memory layout for 2 dimensional arrays, we can determine that in the expression **`multi + row`** as used above, **`multi + row + 1`** must increase value by an amount equal to that needed to "point to" the next row, which in this case would be an amount equal to **`COLS * sizeof(int)`**.

Multidimensional arrays (9/14)

- That says that if the expression `*(*(multi+row)+col)` is to be evaluated correctly at run time, the compiler must generate code which takes into consideration the value of **COLS**, i.e. the 2nd dimension
  - Because of the equivalence of the two forms of expression, this is true whether we are using the pointer expression above, or the array expression `multi[row][col]`
- Thus, to evaluate either expression, a total of **5 values** must be known:
  - The address of the first element of the array, which is returned by the expression `multi`, i.e., the name of the array
  - The size of the type of the elements of the array, in this case `sizeof(int)`.
  - The 2nd dimension of the array, `COLS` in this case
    - **Note: The 1st dimension of the array, `ROWS` in this case, need NOT be known!**
    - **You are just not allowed to exceed it on access…**
  - The specific index value for the first dimension, `row` in this case
  - The specific index value for the second dimension, `col` in this case
- Given all of that, consider the problem of designing a function to manipulate the element values of a previously declared array. For example, one which would set all the elements of the array `multi` to 1.

Multidimensional arrays (10/14)

- So, what is an appropriate function prototype for a function that receives and initializes a 2-dimensional array of integers to a given value?

```
void init_array(int *array, int value);

void init_array(int **array, int value);

void init_array(int array[ ][ ], int value);

void init_array(int array[<rows>][ <columns>], int value);
```

… ?

Multidimensional arrays (11/14) – multidim_2.c

- One possible solution

```c
#include <stdio.h>
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];
void init_array(int array[][COLS], int value);

int main(void) {
    int row, col;
    init_array(multi, 1);
    for (row = 0; row < ROWS; row++) {
        for (col = 0; col < COLS; col++) {
            printf("%d ", multi[row][col]);
        }
        printf("\n");
    }
    return 0;
}
void init_array(int array[][COLS], int value) {
    int row, col;
    for (row = 0; row < ROWS; row++) {
        for (col = 0; col < COLS; col++) {
            array[row][col] = value;
        }
    }
}
```

Multidimensional arrays (12/14)

- Within the function **`init_array`** in the previous solution, we have used the values **`#defined`** by **`ROWS`** and **`COLS`** that set the limits on the for loops
- These **`#defines`** are just constants as far as the compiler is concerned, i.e. there is nothing to connect them to the array size within the function
- **`row`** and **`col`** are local variables, of course

- The **formal parameter definition** **`int array[][COLS]`** permits the compiler to determine the characteristics associated with the pointer value that will be passed at run time
  - » We really don't need the first dimension and, as will be seen later, there are occasions where we would prefer not to define it within the parameter definition, out of habit or consistency
  - » But, the second dimension **must** be used as has been shown in the expression for the parameter. The reason is that we need this in the evaluation of **`m_array[row][col]`** as has been described.

Multidimensional arrays (13/14)

- While the parameter type of **`init_array`** defines the data type (**`int`** in this case) and the automatic variables for row and column are defined in the for loops, only one value can be passed using a single parameter.
  - » In this case, that is the value of **`multi`** as noted in the call statement, i.e. the address of the first element, often referred to as a pointer to the array

- Thus, the only way we have of informing the compiler of the 2nd dimension is by explicitly including it in the parameter definition

- In general, **all dimensions of higher order than one are needed** when dealing with multi-dimensional arrays
  - That is if we are talking about 3-dimensional arrays, the 2nd and 3rd dimension must be specified in the parameter definition

- Does this mean that we cannot write “generic” array manipulation functions?

Multidimensional arrays (14/14) – multidim_3.c

- Thankfully not: pointer arithmetic to the rescue!

```c
#include <stdio.h>
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];
void init_array(int *array, int num_rows, int num_cols, int value);

int main(void) {
    int row, col;
    init_array((int*)multi, ROWS, COLS, 1);
    for (row = 0; row < ROWS; row++) {
        for (col = 0; col < COLS; col++) {
            printf("%d ",multi[row][col]);
        }
        printf("\n");
    }
    return 0;
}
void init_array(int *array, int num_rows, int num_cols, int value) {
    int row, col;
    for (row = 0; row < num_rows; row++) {
        for (col = 0; col < num_cols; col++) {
            *(array + (row * num_cols) + col) = value;
        }
    }
}
```

Pointers to arrays (1/5)

- Pointers can be "pointed at" any type of data object, including arrays
- It is important to understand how we do this when it comes to multi-dimensional arrays

- Before, we stated that given an array of integers we could point an integer pointer at that array using:

```
int *ptr;
ptr = &my_array[0];
```

- The type of the pointer variable must, of course, match the type of the first element of the array
- In addition, we can use a pointer as a formal parameter of a function which is designed to manipulate an array

Pointers to arrays (2/5)

- Given, e.g.:

```
int array[3] = {'1', '5', '7'};
void a_func(int *p);
```

- Some programmers might prefer to write the function prototype as:

```
void a_func(int p[]);
```

  which would tend to inform others who might use this function that the function is designed to manipulate the elements of an array

- In either case, what actually gets passed is the value of a pointer to the first element of the array, independent of which notation is used in the function prototype or definition

Pointers to arrays (3/5)

- We now turn to the problem of the 2-dimensional array. As stated earlier, C interprets a 2-dimensional array as an array of one-dimensional arrays.
  - » That being the case, the first element of a 2-dimensional array of integers is a one-dimensional array of integers
  - » A pointer to a two-dimensional array of integers must be a pointer to that data type

- One way of accomplishing this is through the use of the keyword **`typedef`**
- **typedef** assigns a new name to a specified data type

- For example:

```
typedef unsigned char byte;
```

causes the name **`byte`** to mean type **`unsigned char`**
Hence **`byte b[10];`** would create an array of unsigned characters

Pointers to arrays (4/5)

- Now, while using typedef makes things clearer for the reader and easier on the programmer, it is not really necessary. What we need is a way of declaring a pointer like p1d without the need of the typedef keyword.

- It turns out that this can be done and that

  ```
  int (*p1d)[10];
  ```

  is the proper declaration, i.e. p1d here is a pointer to an array of 10 integers just as it was under the declaration using the array type
  → One pointer (to an array) + 10 integers

- Note that this is **different** from

  ```
  int *p1d[10];
  ```

  which would make p1d the name of an array of 10 pointers to type int
  → 10 pointers to integers

Pointers to arrays (5/5)

- Now consider this type definition:

  `typedef int Array[10];`

- **`Array`** becomes a data type for an array of 10 integers. i.e. **`Array my_arr;`** declares **`my_arr`** as an array of 10 integers and **`Array arr2d[5];`** makes **`arr2d`** an array of 5 arrays of 10 integers each

- Note that **`Array *p1d;`** makes p1d a pointer to an array of 10 integers
  - Because *p1d points to the same type as arr2d, assigning the address of the two-dimensional array **`arr2d`** to **`p1d`**, the pointer to a one-dimensional array of 10 integers is acceptable
  - I.e.: **`p1d = &arr2d[0];`** or **`p1d = arr2d;`** are both correct.

- Since the data type we use for our pointer is an array of 10 integers we would expect, that incrementing **`p1d`** by 1 would change its value by **`10*sizeof(int)`**, which it does. That is, **`sizeof(*p1d)`** is **40** (in IA32).

Pointers and dynamic allocation of memory (1/16)

- There are times when it is convenient to allocate memory at run time using **`malloc()`**, **`calloc()`**, or other allocation functions
  - » Using this approach permits postponing the decision on the size of the memory block need to store an array, for example, until run time
  - » It also permits using a section of memory for the storage of an array of integers at one point in time, and then when that memory is no longer needed it can be freed up for other uses, such as the storage of an array of structures

```
int *array = (int *)calloc(N, sizeof(int));
array[0] = 15;
```

- When memory is allocated, the allocating function returns a pointer. The type of this pointer is **void ***.
- Since a void pointer can be assigned to a pointer variable of any object type, an (int *) cast is not needed – but **highly** recommended!
- **`void *malloc(size_t size)`**: Reserves **`size`** bytes
- **`void *calloc(size_t num, size_t size)`**: Res. **`num*size`** bytes
  - And initializes all bits to zero
- Both functions return **`NULL`** if the memory could not be allocated

Pointers and dynamic allocation of memory (2/16)

- Even with a reasonably good understanding of pointers and arrays, one place the newcomer to C is likely to stumble at first is in the **dynamic allocation of multidimensional arrays**

- In general, we would like to be able to access elements of such arrays using array notation, not pointer notation, wherever possible. However, depending on the application we may or may not know both dimensions at compile time. This leads to a variety of ways to go about our task.
  - As we have seen, when dynamically allocating a one-dimensional array its dimension can be determined at run time
  - Now, when using dynamic allocation of higher order arrays, we never need to know the first dimension at compile time
  - Whether we need to know the higher dimensions depends on how we go about writing the code

- We will discuss next various methods of dynamically allocating room for 2-dimensional arrays of integers

Pointers and dynamic allocation of memory (3/16)

Method 1

- One way of dealing with the problem is through the use of the `typedef` keyword. To allocate a 2-dimensional array of integers, recall that the following two notations result in the same object code being generated:

```
multi[row][col] = 1;          *(*(multi + row) + col) = 1;
```

- It is also true that the following two notations generate the same code:

```
multi[row]                *(multi + row)
```

- Since the one on the right must evaluate to a pointer, the array notation on the left must also evaluate to a pointer
  - » Actually, `multi[n]` evaluates to a pointer to that array of integers that make up the n-th row of our 2-dimensional array. That is, multi can be thought of as an array of arrays and multi[n] as a pointer to the n-th array of this array of arrays (here the word pointer is being used to represent an address value).

- When reading such statements one must be careful to distinguish between the constant address of an array and a variable pointer which is a data object in itself

Pointers and dynamic allocation of memory (4/16) – method1.c

Method 1

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

#define COLS 5

typedef int RowArray[COLS];

RowArray *rptr;

int main(void) {
    int nrows = 10;
    int row, col;

    rptr = (RowArray *) malloc(nrows * COLS * sizeof(int));

    for (row = 0; row < nrows; row++) {
        for (col = 0; col < COLS; col++) {
            rptr[row][col] = 1;
        }
    }
    return 0;
}
```

Pointers and dynamic allocation of memory (5/16)

Method 2

- In Method 1, **rptr** turned out to be a pointer to type "one dimensional array of **COLS** integers"

- Without the need of typedef, we can write:
  ```
  int (*xptr)[COLS];
  ```
  the variable **xptr** will have all the same characteristics as the variable rptr in Method 1 above

- Here xptr is a pointer to an array of integers and the size of that array is given by the #defined COLS

Pointers and dynamic allocation of memory (6/16) – method2.c

Method 2

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

#define COLS 5

int (*xptr)[COLS];

int main(void) {
    int nrows = 10;
    int row, col;

    xptr = (int(*)[]) malloc(nrows * COLS * sizeof(int));

    for (row = 0; row < nrows; row++) {
        for (col = 0; col < COLS; col++) {
            xptr[row][col] = 1;
        }
    }
    return 0;
}
```

Pointers and dynamic allocation of memory (7/16)

Method 3

- Consider the case where we do not know the number of elements in each row at compile time, i.e. both the number of rows and number of columns must be determined at run time

- One way of doing this would be to create an array of pointers to type int and then allocate space for each row and point these pointers at each row

Pointers and dynamic allocation of memory (8/16) – method3.c

Method 3

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

int main(void) {
    int nrows = 5; /* Both nrows and ncols could be evaluated */
    int ncols = 10; /* or read in at run time */
    int row;
    int **rowptr;
    rowptr = (int**) malloc(nrows * sizeof(int *));
    if (rowptr == NULL) {
        puts("Failure to allocate room for row pointers.\n");
        exit(0);
    }
    printf("Index Pointer(hex) Pointer(dec) Diff.(dec)");
    for (row = 0; row < nrows; row++) {
        rowptr[row] = (int*) malloc(ncols * sizeof(int));
        if (rowptr[row] == NULL) {
            printf("\nFailure to allocate for row[%d]\n", row);
            exit(0);
        }
        printf("\n%5d %12p %12lu", row, rowptr[row], (long) rowptr[row]);
        if (row > 0)
            printf(" %9d",(int)(rowptr[row] - rowptr[row-1]));
}
// Cleanup omitted here on slides!
    return 0;
}
```

Pointers and dynamic allocation of memory (9/16)

Method 3

Example output:

```
Index Pointer(hex) Pointer(dec) Diff.(dec)
    0     0xa040848    168036424
    1     0xa040878    168036472          12
    2     0xa0408a8    168036520          12
    3     0xa0408d8    168036568          12
    4     0xa040908    168036616          12
```

Pointers and dynamic allocation of memory (10/16)

Method 3

- In the above code **`rowptr`** is a pointer to pointer to type int. In this case it points to the first element of an array of pointers to type int. Consider the number of calls to malloc():

To get the array of pointers	1 call
To get space for the rows	5 calls

Total	6 calls

- If you choose to use this approach note that while you can use the array notation to access individual elements of the array, it **does not mean that the data in the "two-dimensional array" is contiguous in memory**.
  - » You can, however, use the array notation to access the data just as if it were a contiguous block of memory. For example, you can write:

```
rowptr[row][col] = 176;
```

    just as if **`rowptr`** were the name of a two-dimensional array created at compile time. Of course **`row`** and **`col`** must be within the bounds of the array you have created, just as with an array created at compile time.
  - » Also, each row could contain a different number of columns (but DON'T do this!)

Pointers and dynamic allocation of memory (11/16)

Method 4

- In this method we allocate a **single block** of memory to hold the **whole** array first
- We then create an array of pointers to point to each row
  - Thus even though the array of pointers is being used, the actual data in memory is contiguous

Pointers and dynamic allocation of memory (12/16) – method4.c

Method 4

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

int main(void) {
    int **rptr;
    int *aptr;
    int *testptr;
    int k;
    int nrows = 5; /* Both nrows and ncols could be evaluated */
    int ncols = 10; /* or read in at run time */
    int row, col;
    /* we now allocate the memory for the array */
    aptr = (int *) malloc(nrows * ncols * sizeof(int));
    if (aptr == NULL) {
        puts("Failure to allocate room for the array");
        exit(0);
    }
    /* next we allocate room for the pointers to the rows */
    rptr = (int **) malloc(nrows * sizeof(int *));
    if (rptr == NULL) {
        puts("Failure to allocate room for pointers");
        exit(0);
    }
    /* and now we 'point' the pointers */
    for (k = 0; k < nrows; k++) {
        rptr[k] = aptr + (k * ncols);
    }

    ...
```

Pointers and dynamic allocation of memory (13/16)

Method 4

```
    ...
    /* Now we illustrate how the row pointers are incremented */
    printf("Illustrating how row pointers are incremented\n");
    printf("\nIndex Pointer(hex) Diff.(dec)");
    for (row = 0; row < nrows; row++) {
        printf("\n%d %p", row, rptr[row]);
        if (row > 0)
            printf(" %d", (int) (rptr[row] - rptr[row-1]));
    }
    printf("\n\nAnd now we print out the array\n");
    for (row = 0; row < nrows; row++) {
        for (col = 0; col < ncols; col++) {
            rptr[row][col] = row + col;
            printf("%d ", rptr[row][col]);
        }
        putchar('\n');
    }
    puts("\n");
    /* and here we illustrate that we are, in fact, dealing with
    a 2-dimensional array in a contiguous block of memory. */
    printf("And now we demonstrate that they are contiguous in memory\n");
    testptr = aptr;
    for (row = 0; row < nrows; row++) {
        for (col = 0; col < ncols; col++)
            printf("%d ", *(testptr++));
        }
        putchar('\n');
    }
    return 0;
}
```

Pointers and dynamic allocation of memory (14/16)

Method 4

Example output:

```
Illustrating how row pointers are incremented

Index Pointer(hex) Diff.(dec)
    0   0xa040428
    1   0xa040450        10
    2   0xa040478        10
    3   0xa0404a0        10
    4   0xa0404c8        10

And now we print out the array
 0  1  2  3  4  5  6  7  8  9
 1  2  3  4  5  6  7  8  9 10
 2  3  4  5  6  7  8  9 10 11
 3  4  5  6  7  8  9 10 11 12
 4  5  6  7  8  9 10 11 12 13


And now we demonstrate that they are contiguous in memory
 0  1  2  3  4  5  6  7  8  9
 1  2  3  4  5  6  7  8  9 10
 2  3  4  5  6  7  8  9 10 11
 3  4  5  6  7  8  9 10 11 12
 4  5  6  7  8  9 10 11 12 13
```

Pointers and dynamic allocation of memory (15/16)

- Let’s look at again at the number of calls to malloc():

To get room for the array itself	1 call
To get room for the array of ptrs	1 call

Total	2 calls

- Some additional details:
  - » Each call to malloc() creates additional space overhead, since malloc() is generally implemented by the operating system forming a linked list which contains data concerning the size of the block
  - » More importantly, with large arrays (several hundred rows) keeping track of what needs to be freed when the time comes can be more cumbersome
  - » This, combined with the contiguousness of the data block that permits initialization to all zeroes using **memset()** would seem to make the second alternative the preferred one
- Why is the difference between rows only 10 (i.e. what “units” are these)?
  - » Take a look what pointers we subtract (pointer arithmetic ≠ integer arithmetic)!
- Something to try at home:
  - » Dynamic allocation of a three-dimensional array!

Pointers and dynamic allocation of memory (16/16)

- Summary of the four methods

	# columns must be known	# rows must be known	# calls required for (de)allocation	contiguous memory block
Method 1 `(RowArray*)`	YES	NO	1	YES
Method 2 `(int(*)[])`	YES	NO	1	YES
Method 3 `(int**) & (int*)`	NO	NO	Rows + 1	NO
Method 4 `(int*)`	NO	NO	2	YES