

Arrays 2: Functions and Pointers

PHYS2G03

© James Wadsley,
McMaster University

Array Types and Declarations

- Arrays can be of any type that a scalar variable can be – including structs and classes
- They must be declared in the same way as the scalar version the only difference being that a size for the arrays is specified using []

Arrays of struct

```
struct point { float x,y; };  
const int N = 100;  
point q; // a single point  
point p[N]; // an array of N points  
  
p[0].x = 0.0; // first point  
p[0].y = 2.0;  
p[1].x = 10.0; // second point  
p[1].y = 2.0;  
  
for (i=2;i<100;i++) { // do the other 98!  
    p[i].x = p[i-1].x+10.0;  
    p[i].y = p[i-1].y;  
}
```

Arrays in memory

- Arrays are just handled as a big piece of memory
- The code keeps track of where it starts in memory using a **pointer**
- It finds individual elements by offsets from the starting point
- Pointers are just long integers describing a place in memory (8 bytes long typically)

Statement	Memory Contains
float a[4];	a₀
	a₁
	a₂
	a₃

Making and using pointers

You can ask where a variable is stored in memory

float b;

&b “Address of” b – where b is stored

&b is a pointer to a float

float a[1]; // a is an array of 1 float

a is a pointer to that float

*a “contents of” a
 – the real number in that memory

*a is equivalent to a[0] They can both be used as a float

Address of (variable)

& “address of” turns a variable into a pointer
pointer = memory address

float x; // a float

&x a pointer to memory, just like an array

(&x)[0] the first float – x again!

(&x)[1] memory error– there is no second float!

as noted earlier this error is not detected
for you – it just changes data unpredictably

Contents of (pointer)

* “contents of” turns pointers into regular variables

If name is a pointer (or an array),

*name is equivalent to name[0]

float y[2]; // an array of two floats

*y = 2.0; // A float – the same as y[0]

y[0] = 2.0; // Equivalent

Pointer operators

& “address of” undoes * “contents of”

float x; // a float

float y[2]; // an array of two floats

x = .01;

*(&x) = 0.01 ; // equivalent to x=.01

y[0] = 2.0;

*(&y[0])) = 2.0; // equivalent to y[0]=2

&(*y[1])) = 6.3; // not legal since it is a pointer

Why Pointers?

Pointers let you directly change variables. This is particularly useful in a function

- Regular function arguments are only copies of variables
- If you send a pointer, the function can change the original variable

Regular Functions

```
void FunctionByValue( float b )
{
    // Initially b = 2
    b =100;
    // Now b = 100
    return; // Now b is gone, a pointless function
}
```

The function only gets the value of b from the outside

```
int main() {
    float A;
    A=2;
    // Here A = 2
    FunctionByValue( A );
    // Here A = 2 still!
```

When a regular function is called, only a copy of the arguments is sent along

Regular Functions

```
void FunctionByValue( float b )  
{  
    // b = 200 first time, 84 second time  
    b =100;  
    // Now b = 100  
    return; // Now b is gone  
}
```

```
int main() {  
    FunctionByValue( 200 )  
  
    FunctionByValue( 10+2*37 );  
}
```

The function only gets the value of b from the outside

When a regular function is called, only a copy of the arguments is sent along. This means any integer expression is legit as an argument

Regular Functions

```
void FunctionByValue( float b )
{
    // b = 200 first time, 84 second time
    b =100;
    // Now b = 100
    return; // Now b is gone
}
```

```
int main() {
    float A;
    A=2;
    // Here A = 2
    FunctionByValue( A );
    // Here A = 2 still!
```

The function only gets the value of b from the outside

When a regular function is called, only a copy of the arguments is sent along. This means the original variable A is safe and can't be changed

Pointer Arguments to Functions

```
void FunctionByPointer( float *pb )
{
    // pb is a pointer
    // *pb is a float
    *pb =100;
    // Now *pb = 100 so A = 100
}
```

```
int main() {
    float A;
    A=2;
    // Here A = 2

    FunctionByPointer( &A );
    // Here A = 100
}
```

`float *pb` means a pointer to a float. The function now gets a memory location from the outside. It can change outside variables!

Now we must send a pointer to a float
For this we use address of, `&A`
`A` can now be changed!

Pointer Arguments to Functions

```
void FunctionByPointer( float *pb )  
{  
    // pb is a pointer  
    // *pb is a float  
    *pb = 100;  
    // Now *pb = 100  
}
```

```
int main() {  
    FunctionByPointer( &(amp;10+2*37) );  
}
```

This code is junk!

An arbitrary expression
does not have a
memory location
Only variables do

Will not compile

C++ references: Direct access to memory for arguments to Functions

```
void FunctionByReference( float &b )  
{  
    // b is a reference  
    b =100;  
    // Now b = 100 so A = 100  
}
```

```
int main() {  
    float A;  
    A=2;  
    // Here A = 2  
  
    FunctionByReference( A );  
    // Here A = 100
```

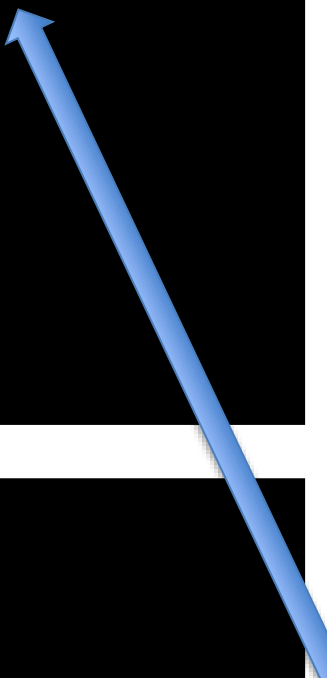
By using a reference
float &b
the function now gets
a memory location just
as if it were a pointer

Using references you
need to be careful and
realize it will change
the original

Give-away is use of &
in the function
declaration

C++ const references: Read-only access to memory for arguments

```
void FunctionByConstReference( const float  
    &b )  
{  
    // b is a reference  
    b =100; // illegal!  
}
```



```
int main() {  
    float A;  
    A=2;  
    // Here A = 2  
  
    FunctionByConstReference( A );  
    // Here A = 2
```

By using a constant reference
`const float &b`
We are no longer allowed to change `b`!
(somewhat pointless in this case)

Function has direct access. But from the declaration we know `A` is safe from changes

C++ references: Direct access to memory for arguments to Functions

```
void FunctionByReference( float &b )  
{  
    // b is a reference  
    b =100;  
    // Now b = 100 so A = 100  
}
```

```
int main() {  
    float A;  
    A=2;  
    // Here A = 2  
  
    FunctionByReference( A );  
    // Here A = 100
```

References are considered safer than pointers because they must point at valid memory containing a variable of the kind declared (here b MUST access a float variable). We can also control that access using const

C++ Functions: 3 options

```
void FunctionByValue(  
    float b )
```

```
{  
    b=100;  
}
```

```
void FunctionByPointer(  
    float *pb )
```

```
{  
    *pb=100;  
}
```

```
void FunctionByReference(  
    float &b )
```

```
{  
    b=100;  
}
```

```
int main() {  
    float A;  
    A=2;  
    // Here A = 2  
  
    FunctionByValue( A );  
    // Here A = 2
```

```
int main() {  
    float A;  
    A=2;  
    // Here A = 2  
  
    FunctionByPointer( &A );  
    // Here A = 100
```

```
int main() {  
    float A;  
    A=2;  
    // Here A = 2  
  
    FunctionByReference( A );  
    // Here A = 100
```

Arrays and functions

- When you make an array an argument to a function – you are sending the memory location of the first element
- Using pointer arithmetic, the function can access the whole array by stepping in memory to the next elements
- Because you get the memory location you can change the values

Arrays to Functions

```
void SomeKindaSort( float b[], int n )
{
    std::cout << "I am sorting this list: \n";

    for (i=0;i<n;i++) std::cout << b[i];
    ...
}
```

```
int main() {
    float A[100];

    ...

    SomeKindaSort( A, 100 );
}
```

You can use:

float name[] OR

float *name

The function doesn't know the size of the array. So we sent n too

To send an array to a function you don't use the []

Without the [], A is a pointer to the memory where A[0] is stored

Arrays to Functions

```
void SomeKindaSort( float *b, int n )
{
    std::cout << "I am sorting this list: \n";

    for (i=0;i<n;i++) std::cout << b[i];
    ...
}
```

```
int main() {
    float A[100];

    ...

    SomeKindaSort( A, 100 );
}
```

You can use:

float name[] OR

float *name

The function doesn't know the size of the array. So we sent n too

To send an array to a function you don't use the []

Without the [], A is a pointer to the memory where A[0] is stored

Pointers in Functions

```
void SomeKindaSort( float *b, int n )
{
    std::cout << "I am sorting this list: \n";

    for (i=0;i<n;i++) std::cout << b[i];
    ...
}
```

```
int main() {
    float A;

    ...

    SomeKindaSort(&A, 1 );
}
```

You can use:

float name[] OR

float *name

The function doesn't know the size of the array. So we sent n

A regular variable is like an array of one thing!
&name turns a regular variable into a pointer to its memory – just like an array

Pointers in Functions

```
void SomeKindaSort( float *b, int n )
{
    std::cout << "I am sorting this list: \n";

    for (i=0;i<n;i++) std::cout << b[i];
    ...
}
```

```
int main() {
    float A;

    ...

    SomeKindaSort(&A, 1 );
}
```

Pointers to floats:

b

&A

&b[0]

These point at memory
where A is stored

floats (all equal A):

b[0]

A

*b

*b is the same as b[0]

Preferred way:

Arrays to Functions

```
void SomeKindaSort( float b[], int n )
{
    std::cout << "I am sorting this list: \n";

    for (i=0;i<n;i++) std::cout << b[i];
    ...
}
```

```
int main() {
    float A[100];

    ...

    SomeKindaSort( A, 100 );
}
```

I prefer this syntax
It is obvious that the
function expects an
array (but doesn't
know the size)

A is the array and we
send it, (not just one
index of it).

Note: because it's a
pointer we CAN
change original values
A[i] at will

Safe approach: preventing changes to Arrays in Functions with const

```
void SomeKindaStats( const float b[], int n )
{
    std::cout << "I am getting stats on this list: \n";

    for (i=0;i<n;i++) std::cout << b[i];
    ...
}
```

```
int main() {
    float A[100];

    ...

    SomeKindaStats( A, 100 );
}
```

Some functions, (like a histogram maker or other statistics) are not meant to change the array. You may want to ensure function can't change your raw data. You can force the function to treat the elements of A[i] as constants it can't change: read only

Arrays in memory

- Arrays are just handled as a big piece of memory
- The code keeps track of where it starts in memory using the **pointer: a**
- It finds individual elements by offsets from the starting point:
a+offset
- This is how the machine code does it in practice. It doesn't know what locations are valid, it just counts from the start

Statement	Memory Contains
float a[4];	a₀
	a₁
	a₂
	a₃

Arrays in memory

- Arrays vs. Pointers
- `a` is a pointer to a float
- `a[0]` is that float

`a[1]` is the next float
`a+1` is the pointer to that float etc ...

Statement	Memory Contains
<code>float a[4];</code>	<code>a₀</code>
	<code>a₁</code>
	<code>a₂</code>
	<code>a₃</code>

Pointer arithmetic

- When you add to a pointer, it moves it forward to the next thing of that type in memory
- A float is 4 bytes
- If `a` is a pointer to 1st float
- `a+1` points 4 bytes ahead, to the 2nd float
- `a+2` points 8 bytes ahead, to the 3rd float

Pointers are memory locations!

Statement a as pointer	Memory Location	Memory Contains
a	10000	a[0]
a+1	10004	a[1]
a+2	10008	a[2]
a+3	10012	a[3]

- A pointer is just a long integer of some kind (compiler dependent) indicating a place in memory where the program stores something

Higher dimensional Arrays

- C/C++ Arrays can be arrays of anything – including arrays
- This is one way to emulate multidimensional arrays – like a matrix

```
float a[10][10]; // 10 arrays of size 10 (total 100 elements)
float b[10];

a[0][0] = 556.2112;
a[0][1] = b[1];
```

2d Arrays – a Matrix

```
float a[3][3]; // 3 arrays of arrays of size 3 (total 9 elements)

for (i=0;i<3;i++) {
    for (j=0;j<3;j++) {
        if (i==j) a[i][j] = 1.0;  else a[i][j] = 0.0;
    }
}
```

- A 3x3 identity matrix using arrays
- All elements are zero except the diagonal

Higher dimensional Arrays

If `b[i]` is a float, what is `b`?

`b` is a pointer to a float

`a[i][j]` is a float, `a[i]` is a pointer to a float

`a` is a pointer to a pointer to a float

```
float a[10][10]; // array of 10 arrays of size 10 (total 100 elements)
float b[10];
a[0][0] = 556.2112;
a[0][1] = b[1];
```


Higher dimensional Arrays

If $b[i]$ is a float, what is b ?

b is a pointer to a float – a memory location where a float lives

$a[i][j]$ is a float, $a[i]$ is a pointer to a float

a is a pointer to a pointer to a float – a memory location where a memory location is stored that has a float in it.

Mostly you can ignore the pointer aspect

Arrays in Memory...

Statement	Memory Contains
<code>int x;</code>	<code>x</code>
<code>int a[3];</code>	<code>a₀</code>
	<code>a₁</code>
	<code>a₂</code>
<code>float b[2][3];</code>	<code>b_{0,0}</code>
	<code>b_{0,1}</code>
	<code>b_{0,2}</code>
	<code>b_{1,0}</code>
	<code>b_{1,1}</code>
	<code>b_{1,2}</code>

Storage

Arrays are given space sequentially
usually in the order declared

As far as the CPU is concerned it is
all just bytes in memory:

scalars, arrays, 2D arrays all just
take up different amounts of
space

Statement	Memory
<code>int x;</code>	<code>x</code>
<code>int a[3];</code>	<code>a₀</code>
	<code>a₁</code>
	<code>a₂</code>
<code>float b[2][3];</code>	<code>b_{0,0}</code>
	<code>b_{0,1}</code>
	<code>b_{0,2}</code>
	<code>b_{1,0}</code>
	<code>b_{1,1}</code>
	<code>b_{1,2}</code>

Storage

In C, the rightmost index steps through memory the fastest.

e.g. `b[0][1]` comes after `b[0][0]`

`b[1][0]` is far from `b[0][0]`

In principle, you don't need to know that – a correct program gives the same answer regardless

Do you need to care?

Statement	Memory
<code>float b[2][3];</code>	<code>b_{0,0}</code>
	<code>b_{0,1}</code>
	<code>b_{0,2}</code>
	<code>b_{1,0}</code>
	<code>b_{1,1}</code>
	<code>b_{1,2}</code>

Efficient Code

```
intr b[2][3], m,n;  
  
for (m=0;m<3;m++) {  
    for (n=0;n<2;n++) {  
        b[n][m] = setup_function(n,m);  
    }  
}
```

Big steps
back & forth

```
intr b[2][3], m,n;  
  
for (n=0;n<2;n++) {  
    for (m=0;m<3;m++) {  
        b[n][m] = setup_function(n,m);  
    }  
}
```

Sequentially,
in order

Statement	Memory
float b[2][3];	b _{0,0}
	b _{0,1}
	b _{0,2}
	b _{1,0}
	b _{1,1}
	b _{1,2}

Consider how t steps
through memory in
each case

The innermost loop
should go over the
rightmost
subscript where
possible