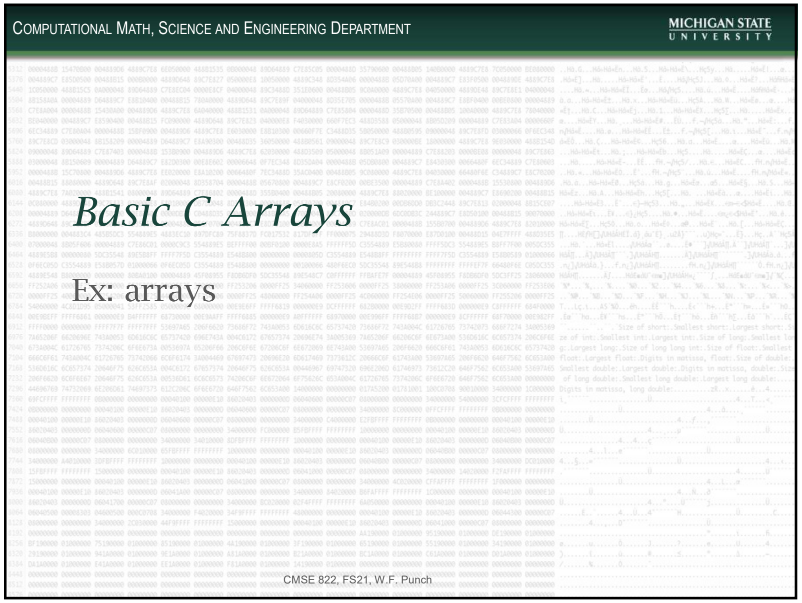
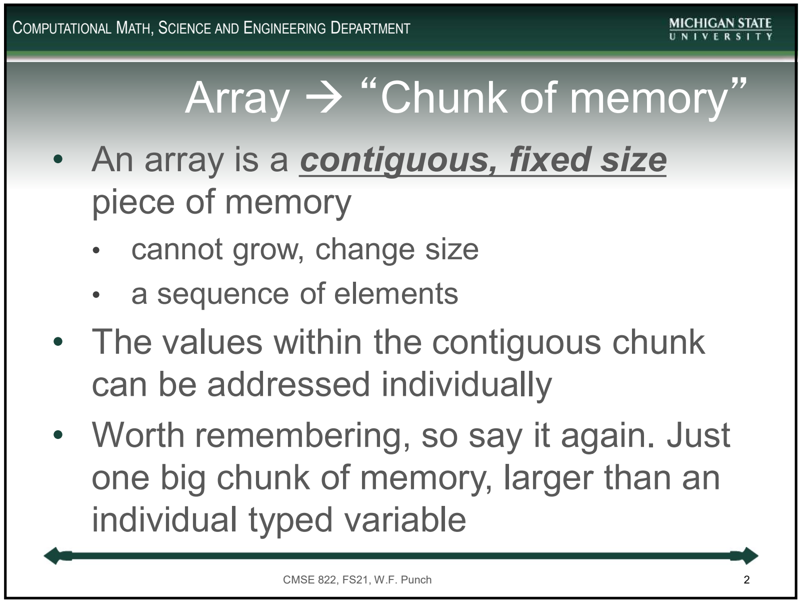


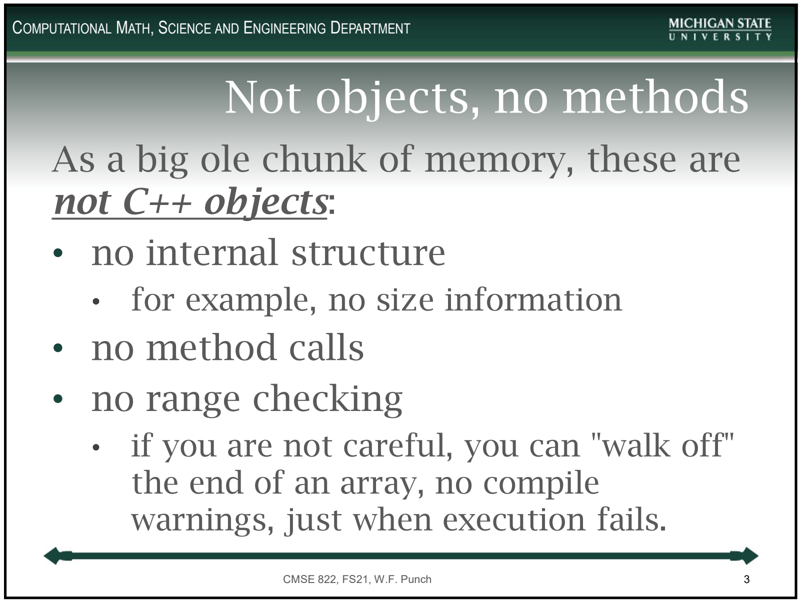
Basic C Arrays

Ex: arrays



Array → "Chunk of memory"

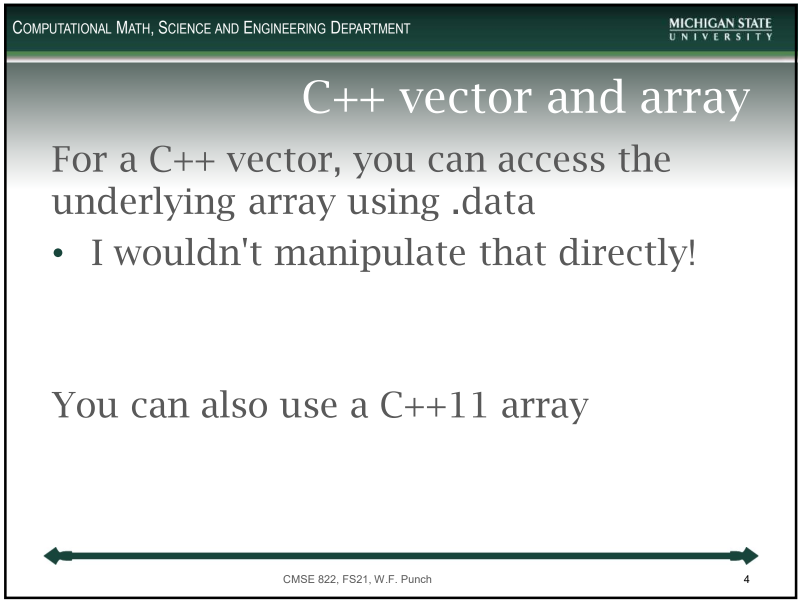
- An array is a ***contiguous, fixed size*** piece of memory
  - cannot grow, change size
  - a sequence of elements
- The values within the contiguous chunk can be addressed individually
- Worth remembering, so say it again. Just one big chunk of memory, larger than an individual typed variable



Not objects, no methods

As a big ole chunk of memory, these are ***not C++ objects***:

- no internal structure
  - for example, no size information
- no method calls
- no range checking
  - if you are not careful, you can "walk off" the end of an array, no compile warnings, just when execution fails.

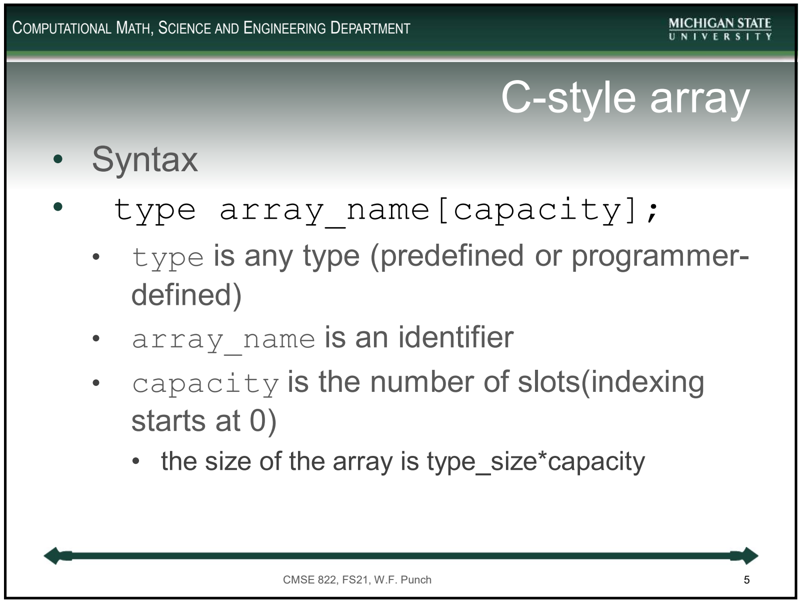


C++ vector and array

For a C++ vector, you can access the underlying array using .data

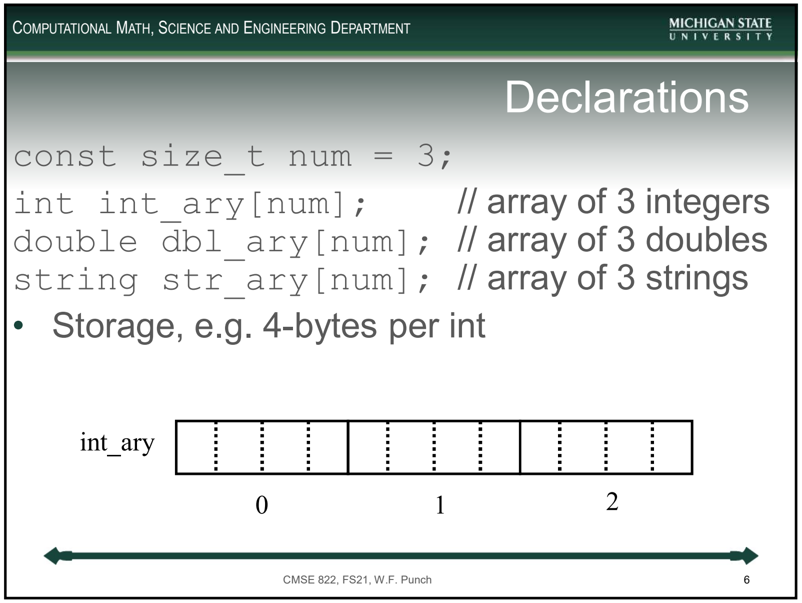
- I wouldn't manipulate that directly!

You can also use a C++11 array



C-style array

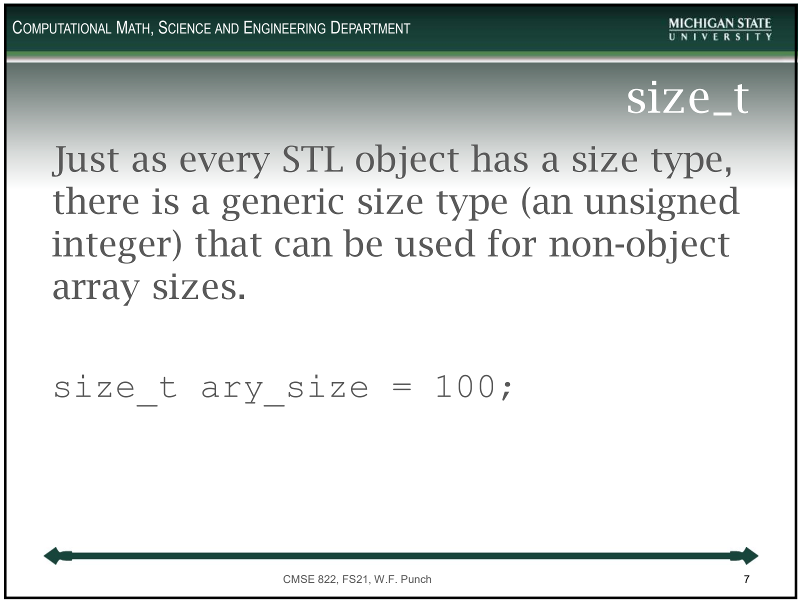
- Syntax
- `type array_name[capacity];`
  - `type` is any type (predefined or programmer-defined)
  - `array_name` is an identifier
  - `capacity` is the number of slots(indexing starts at 0)
    - the size of the array is type_size*capacity



Declarations

```
const size_t num = 3;
int int_ary[num];      // array of 3 integers
double dbl_ary[num];   // array of 3 doubles
string str_ary[num];   // array of 3 strings
```

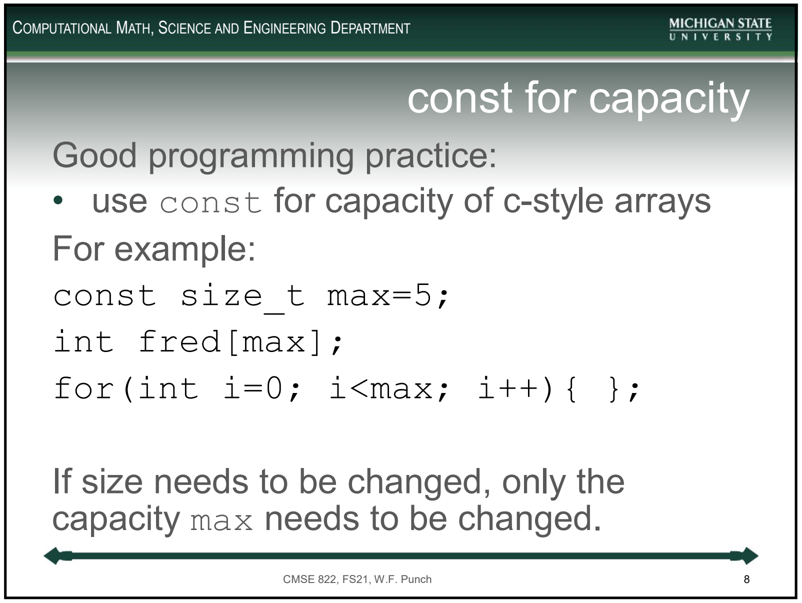
- Storage, e.g. 4-bytes per int



size_t

Just as every STL object has a size type, there is a generic size type (an unsigned integer) that can be used for non-object array sizes.

```
size_t ary_size = 100;
```



const for capacity

Good programming practice:

- use `const` for capacity of c-style arrays

For example:

```
const size_t max=5;
int fred[max];
for(int i=0; i<max; i++){ };
```

If size needs to be changed, only the capacity `max` needs to be changed.

Operations

`int ary[3];` // array of 3 ints

- Subscript:
  - assignment `ary[0]=6;`
- Input/Output:
  - the elements are handled as their types, e.g. `cout << ary[0] << endl;`// int 6
- Arithmetic: `ary[1]= ary[1] + 5;`

Initialization

- Syntax: `int ary[4] = {2,4};`
- Behavior: initialize elements starting with leftmost, i.e. element 0. Remaining elements are initialized to zero.

ary	2	4	0	0
	0	1	2	3

- Compiler can also determine size:

`int ary[]={0,1,2};` // size 3

Type is important

- First, each array needs a type so the size of memory requested can be calculated (number of elements * size of type)
- Because of this, each array can only hold elements of the same type (mostly, there's always a way around these things ☺)

arrays and pointers

"degrading" an array

Ex: aryptrs

array vs pointer

When you have a big chunk of memory of some fixed size, there are really two ways to look at it:

- as an array with some fixed size
  - not stored in the array remember!
- as a pointer to the beginning of the memory chunk.

`int ary[]{2,4,0,0};` → `ary`

	2	4	0	0
	0	1	2	3

You could view `ary` as an `int*` pointer to the first element of the array chunk, that is:

```
*ary == ary[0];
*(ary + 1) == ary[1];
ary++; //don't do that, why???
```

mostly equivalent way to express index

One could view the subscript index as an address offset from the beginning pointer to the array.

Remember, pointer arithmetic is based on "element" math

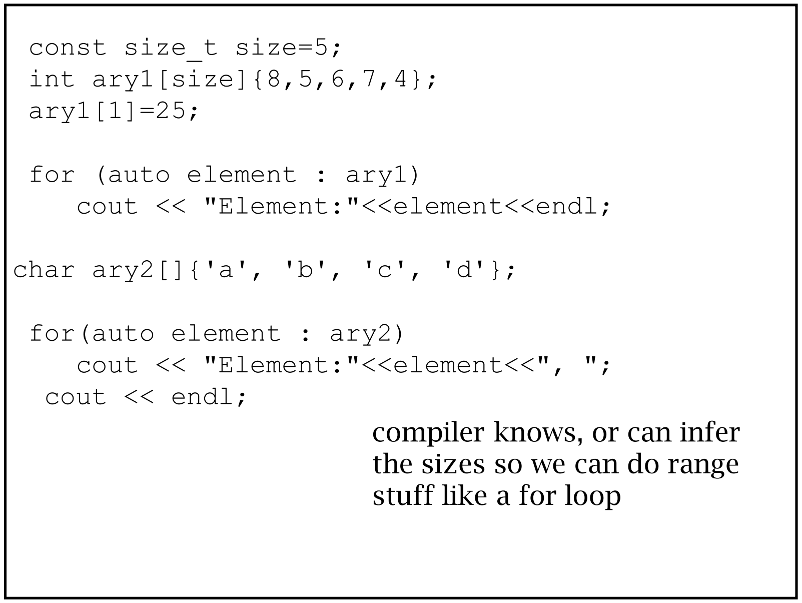
- `ptr+1` points to the next ***value.***
- address goes up by the size of the type to get to the next value

array type vs. pointer type

C++ is sensitive to knowing the size of the array:

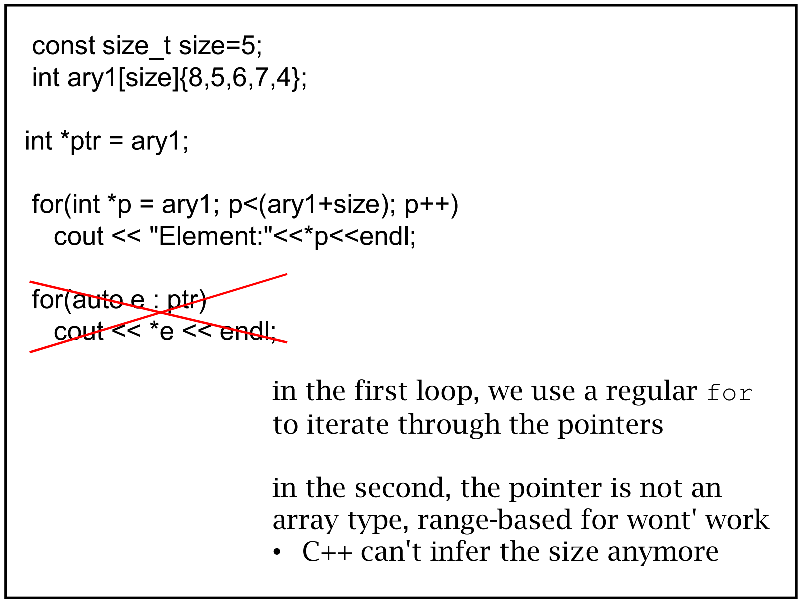
- if the compiler knows the size, then it allows you to do things like range-based for.
- if the compiler cannot know the size, it treats it like a pointer and C++ things won't work

we say, ***degrading*** the array to a pointer



```
 const size_t size=5;
 int ary1[size]{8,5,6,7,4};
 ary1[1]=25;

 for (auto element : ary1)
    cout << "Element:"<<element<<endl;

char ary2[]{'a', 'b', 'c', 'd'};

 for(auto element : ary2)
    cout << "Element:"<<element<<", ";
  cout << endl;
```

compiler knows, or can infer the sizes so we can do range stuff like a for loop



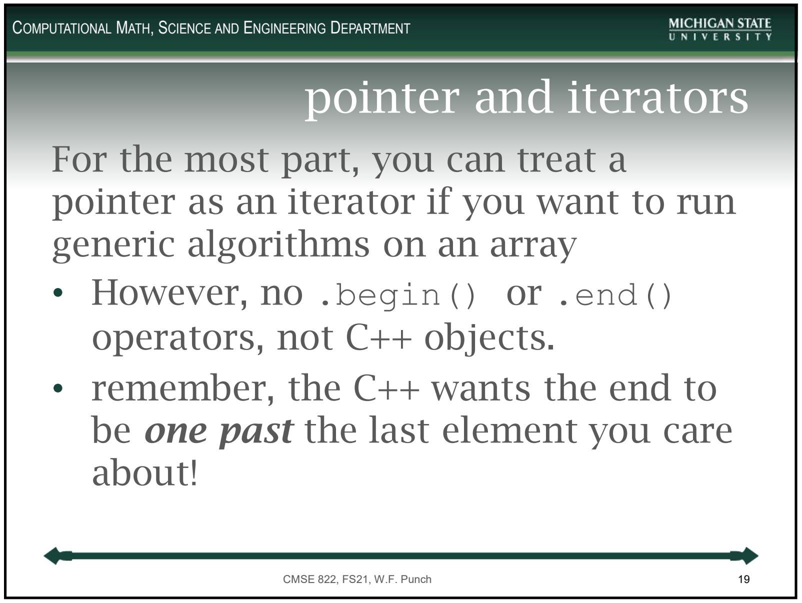
```
 const size_t size=5;
 int ary1[size]{8,5,6,7,4};

int *ptr = ary1;

 for(int *p = ary1; p<(ary1+size); p++)
   cout << "Element:"<<*p<<endl;
```

~~for(auto e : ptr)~~
~~cout << *e << endl;~~

in the first loop, we use a regular `for` to iterate through the pointers

in the second, the pointer is not an array type, range-based for wont' work

- C++ can't infer the size anymore



Computational Math, Science and Engineering Department

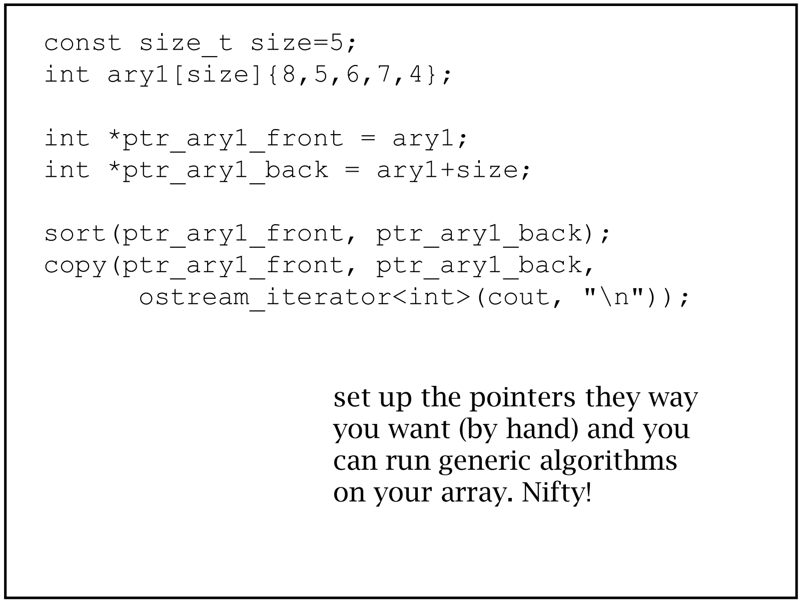
MICHIGAN STATE UNIVERSITY

pointer and iterators

For the most part, you can treat a pointer as an iterator if you want to run generic algorithms on an array

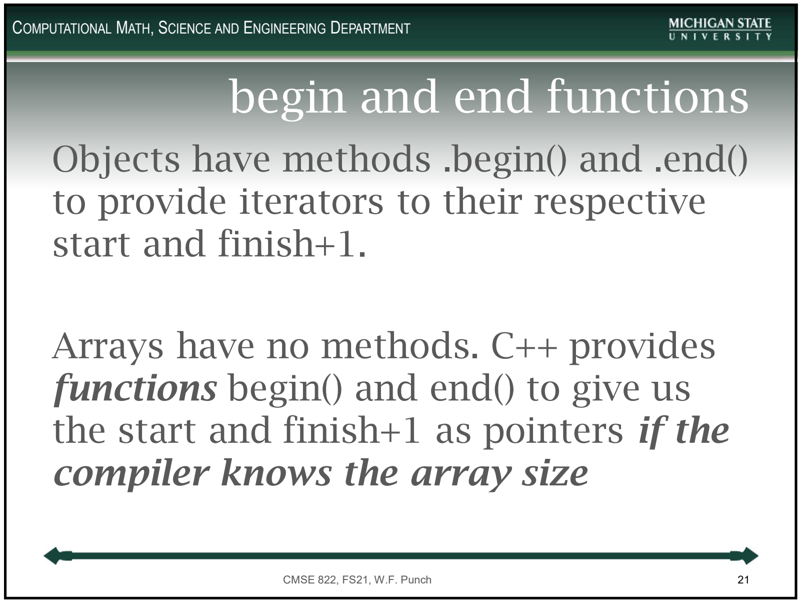
- However, no `.begin()` or `.end()` operators, not C++ objects.
- remember, the C++ wants the end to be ***one past*** the last element you care about!

CMSE 822, FS21, W.F. Punch 19



```
 const size_t size=5;
 int ary1[size]{8,5,6,7,4};

 int *ptr_ary1_front = ary1;
 int *ptr_ary1_back = ary1+size;

 sort(ptr_ary1_front, ptr_ary1_back);
 copy(ptr_ary1_front, ptr_ary1_back,
       ostream_iterator<int>(cout, "\n"));
```

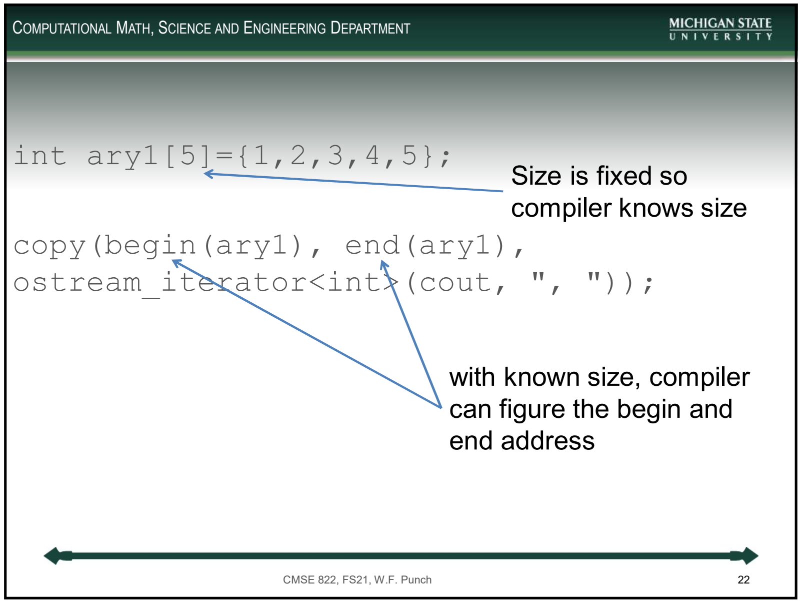
set up the pointers they way you want (by hand) and you can run generic algorithms on your array. Nifty!



begin and end functions

Objects have methods .begin() and .end() to provide iterators to their respective start and finish+1.

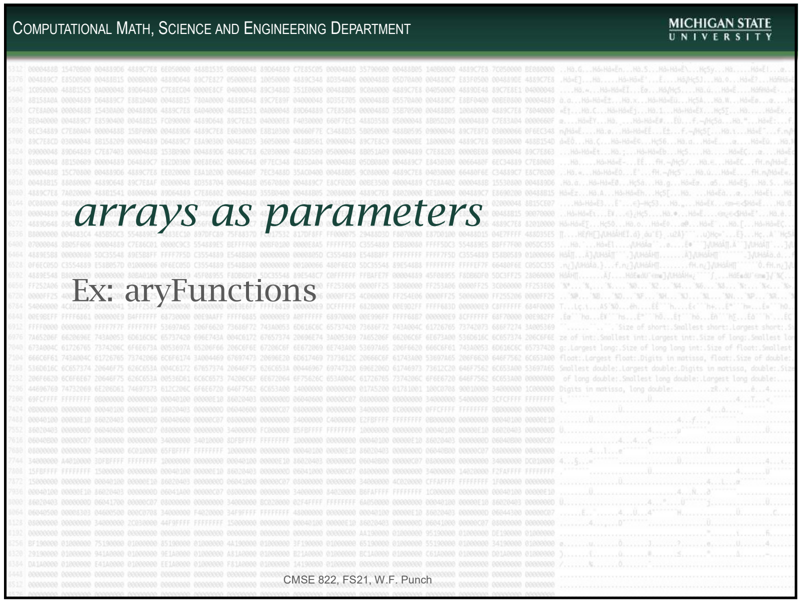
Arrays have no methods. C++ provides ***functions*** begin() and end() to give us the start and finish+1 as pointers ***if the compiler knows the array size***



```
int ary1[5]={1,2,3,4,5};

copy(begin(ary1), end(ary1),
ostream_iterator<int>(cout, ", "));
```

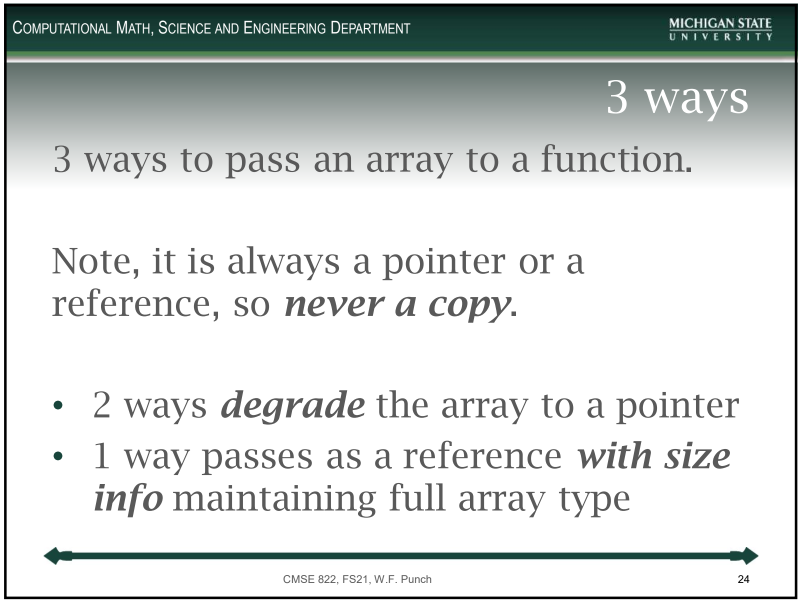
Size is fixed so compiler knows size

with known size, compiler can figure the begin and end address



arrays as parameters

Ex: aryFunctions

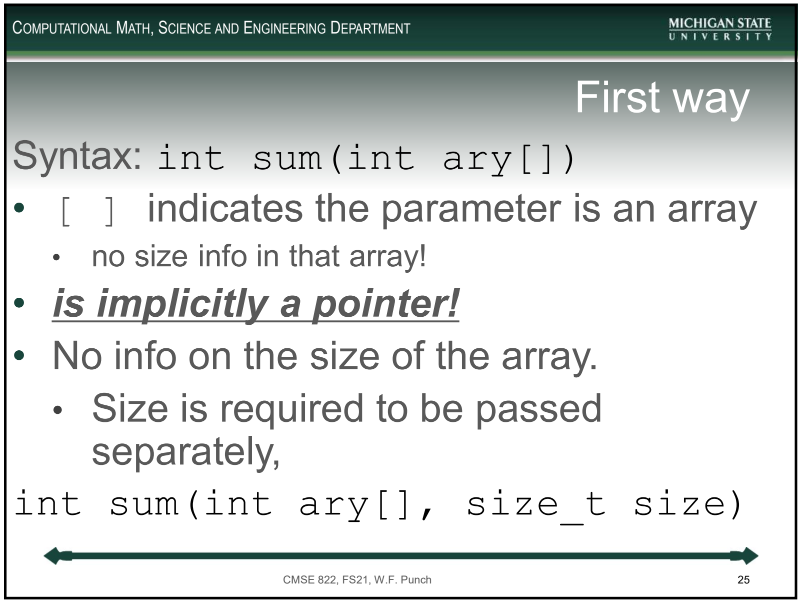


3 ways

3 ways to pass an array to a function.

Note, it is always a pointer or a reference, so ***never a copy***.

- 2 ways ***degrade*** the array to a pointer
- 1 way passes as a reference ***with size info*** maintaining full array type

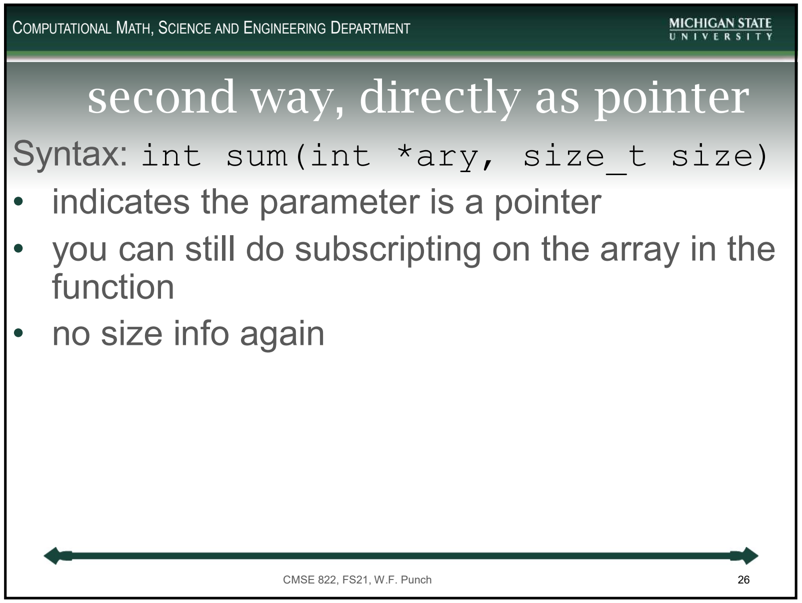


First way

Syntax: `int sum(int ary[])`

- `[ ]` indicates the parameter is an array
  - no size info in that array!
- ***is implicitly a pointer!***
- No info on the size of the array.
  - Size is required to be passed separately,

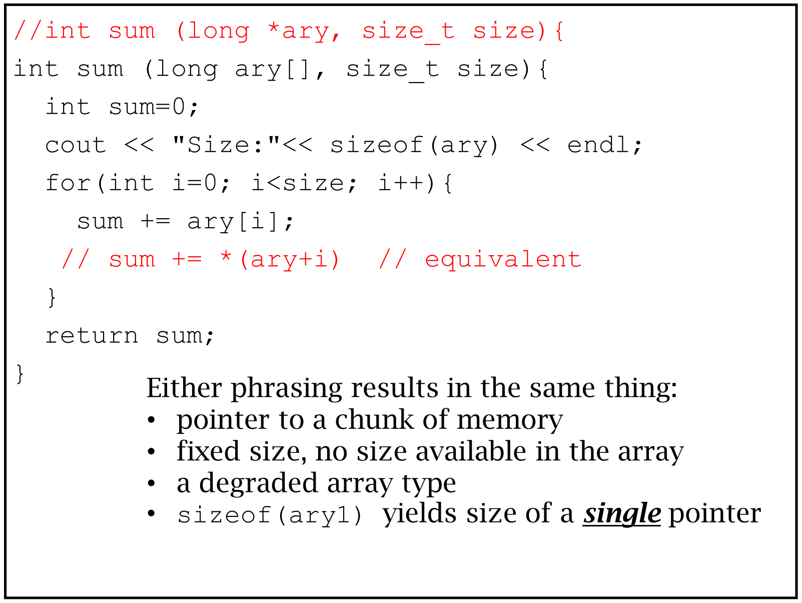
```
int sum(int ary[], size_t size)
```



second way, directly as pointer

Syntax: `int sum(int *ary, size_t size)`

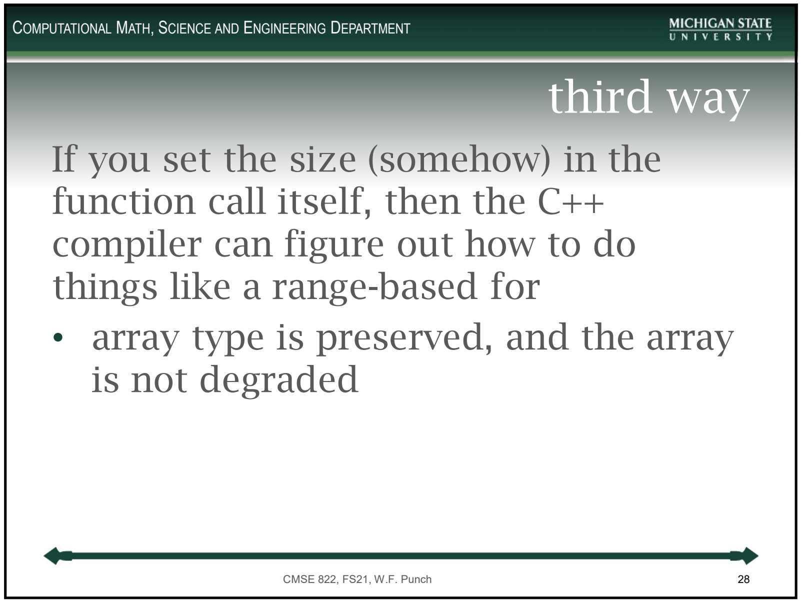
- indicates the parameter is a pointer
- you can still do subscripting on the array in the function
- no size info again



```
//int sum (long *ary, size_t size){
int sum (long ary[], size_t size){
  int sum=0;
  cout << "Size:"<< sizeof(ary) << endl;
  for(int i=0; i<size; i++){
    sum += ary[i];
   // sum += *(ary+i)   // equivalent
  }
  return sum;
}
```

Either phrasing results in the same thing:

- pointer to a chunk of memory
- fixed size, no size available in the array
- a degraded array type
- `sizeof(ary1)` yields size of a ***single*** pointer



third way

If you set the size (somehow) in the function call itself, then the C++ compiler can figure out how to do things like a range-based for

- array type is preserved, and the array is not degraded

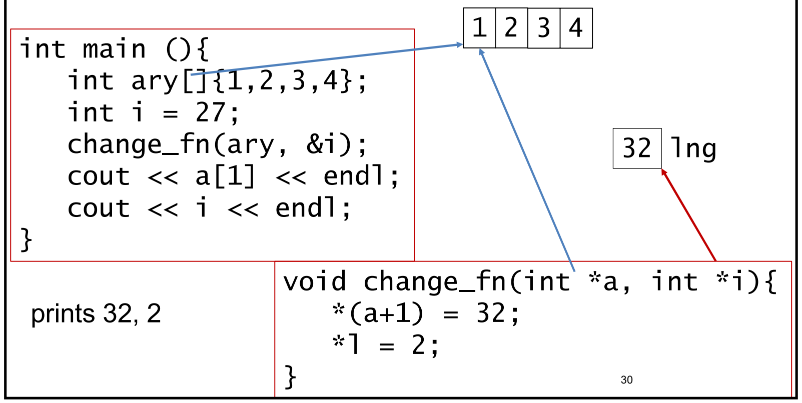
```
long prod(const long (&ary)[3]){
  long result=1;
  cout <<"Size:"<<sizeof(ary)<< endl;

  for(auto &element : ary)
    result = result * element;
  return result;
}

int main ()
   long ary1{1,2,3};
    prod(ary1);
```

size part of parameter, only arrays of length 3.

Some challenging syntax here. Need parens to indicate reference to an array.

- without, it is array of references

If you pass a pointer, then any changes you make in the function are reflected in the main.



```
int main (){
   int ary[]{1,2,3,4};
   int i = 27;
   change_fn(ary, &i);
   cout << a[1] << endl;
   cout << i << endl;
}
```

prints 32, 2

```
void change_fn(int *a, int *i){
   *(a+1) = 32;
   *l = 2;
}
```

30

```
template<typename Type, size_t Size>
long squares(const Type (&ary)[Size]){
  long result=0;
  cout << "Size of info:"<<sizeof(ary)<<endl;
  for(auto element : ary)
    result = result + (element * element);
  return result;
}
```

ask template to deduce the `size_t` of the array, store in var Size, and use as param

Very nice. Allow the compiler to deduce the size (without us setting it explicitly as before) via template, and instantiate the template to new size of each array.

Again, some challenging syntax here!