Arrays and Dynamic Memory

Good Old Fashioned C Arrays

Just like Mom used to program with

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

We can do some C++ things to an array, but it takes some work.

C++11 array vs good ole C array

It is worth noting that C++11 has an object called `array` with equivalent functionality to a C-array

- it is in fact an object, a fixed size sequence
- it has some internal structure
  - it knows its size.

We study C-arrays here

The concept of a C-array is so pervasive that it is worth studying

- one time we don't follow the latest stuff in the C++11 standard

Array

Ex 15.1

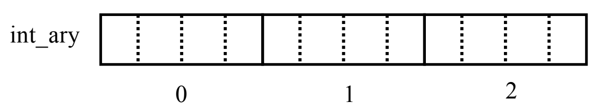
C-style array

- 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



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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;
```

Arrays & Dynamic Memory

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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.

Arrays & Dynamic Memory

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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;`

Arrays & Dynamic Memory

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 15.2

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

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!


```
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

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

arrays as parameters

Ex 15.3

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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

Arrays & Dynamic Memory

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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)
```

Arrays & Dynamic Memory

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

```
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!

dynamic memory

memory on demand
Ex 15.4 and Ex 15.5

compile time vs run time

Good to remind ourselves:

- compile time, what is known at the time of running the compiler to make an executable
- run time, what is known when the user actually runs the executable

STL objects vs us

STL objects know how to get more memory during ***runtime***

- we love them for this. Vectors, Maps, etc. can get bigger when we ask them to as they run

For things like arrays, fixed size non-object:

- they are a fixed size at compile time!

how does the STL do it?

Underlying the STL is the ability to ask for and release memory ***during runtime***.

We can do the same if we wish, but:

- we must be careful. Many (many, many) programmers make mistakes at this point
- ***if the STL can do it, let it***. It is better at it!!!

C++11 or the old way

For once, I want to talk about the "old" way to do dynamic memory, not the latest C++11 way:

- you would rarely see the latest way
- there are some complications that are a little much
- nonetheless, Ex 15.5 uses `shared_ptr` and `unique_ptr` (also look in the book)

so simple, two functions

`new`

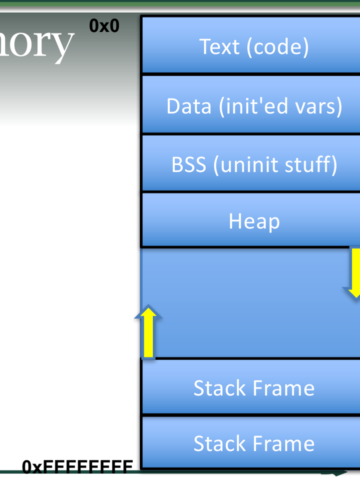
- gets/allocates memory from the OS, returns a ***pointer*** to it (single object or array of those objects)

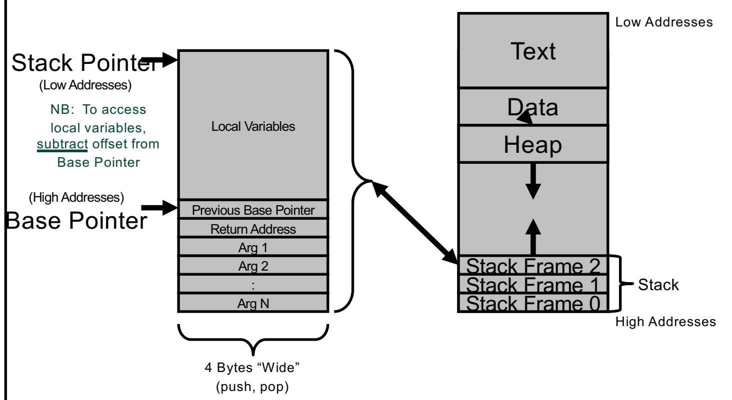
`delete`

- de-allocates the memory, gives it back to the OS.

Process Memory

- Heap
  - Used Explicitly
  - Think `malloc` & `new`
  - Grows Towards Stack
  - May Have “Holes”
  - May Overflow (Run Into Stack)
- Stack
  - Used Implicitly
  - Think `push` & `pop`
  - Grows Towards Heap
  - May Not Have “Holes”
  - May Overflow (Run Into Heap)





ownership of memory

The requests from `new` and `delete` do not change memory in any way, they simply mark a segment of memory as to who "owns" it.

- if you `new` some memory, the OS marks that memory in the heap as ***yours***
- if you never `delete` it, while the program runs the OS thinks it is "gone" i.e., it can't use it.

ownership(2)

if you `delete` some memory, you are simply ceding ownership back to the OS

- the OS is now free to give the memory to some other program
- no contents are ever changed by the OS!! Until the OS gives it to another program and that program changes the memory, that memory looks like how your program left it.

new

`new type (init)`

- allocate new memory of indicated `type`
  - can optionally provide an `init`, not required

or

`new type [size]`

- make `size` elements of type indicated

both return a ***pointer*** to the new memory

delete

`delete ptr;`

- delete (remove ownership) of object pointed to by `ptr`

or

`delete [] ptr;`

- for an array, delete (remove ownership of) all the elements
  - `ptr` points to the beginning of the memory array to be deleted.

constructor call on new memory

You can make a call to a constructor for the new memory, and in this way you can initialize memory

- not required if you will fill the memory yourself
- in general it is a good idea, otherwise the "values" stored in that memory are whatever was left over from the previous user.


```
int main (){
  // basic new
  long *lptr;
  lptr = new long(1234567);
  cout << "lptr:"<<*lptr<<endl;
  delete lptr;
}
```

request memory return pointer

constructor call init memory

delete lptr, cede ownership back to the OS


```
class MyClass{
private:
  long lng_;
  int intgr_;
  string str_;
public:
  MyClass():
     lng_(0), Intgr_(0), str_("X")  {};
  MyClass(long l, int i, string s):
     lng_(l), intgr_(i), str_(s) {};
  friend ostream& operator<<
      (ostream&, const MyClass&);
};
```

```
MyClass *mcptr, *mcptr2;

// default constructor
mcptr = new MyClass;

// 3 param constructor
mcptr2 = new MyClass(123456,
                     123, "Y");

cout << *mcptr;     // 0,0, X
cout << *mcptr2;   // 123456, 123, Y
```

constructor call

non-standard, but g++ allows it

```
size_t size;
cout << "How big:";
cin >> size;
//  not an array type!!
long *ary = new long[size];
fill(ary, size);
dump(cout, ary, size);
delete [] ary;
```


be careful

The prescribed way is to do new and delete.

- the g++ extension is called VLA (variable length arrays)
- not part of C++ (in fact, forbidden). Non-standard so it may not compile elsewhere

memory issues

leaking memory

```
int main (){
  int reps = 2048;
  const size_t chunk = 1048576; // be careful!!!
  long temp = 0;

  for(int i=0;i<reps;i++){
    long *ary = new long[chunk]; // leak!
    ary[0]=0;
    for (int j=1;j<chunk;j++)
      ary[j] = ary[j-1] + temp;
    temp = ary[chunk-1];
  }
}
```

the leak

This is ***leaking memory***

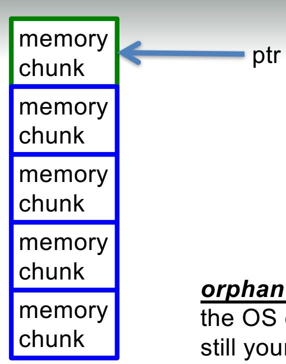
- `new` some memory
  - get a pointer `ptr` to the memory
  - claim ownership from OS
- use the memory (do something)
- reassign `ptr` to some new memory
  - didn't delete the memory `ptr` pointed to
  - can't access that "orphan" memory now
  - OS doesn't know that, still marks it as yours

bottom line

Your program, while it runs, you accumulate ownership of memory from the OS, deleting memory resources.

- the memory footprint of your running program grows
  - uselessly, you aren't using that memory
  - even if you wanted to, you lost pointer to the memory. It is orphaned
  - OS doesn't know, can't reuse that memory.

for (int i=0; i<10; ++i) ptr = new long[10];

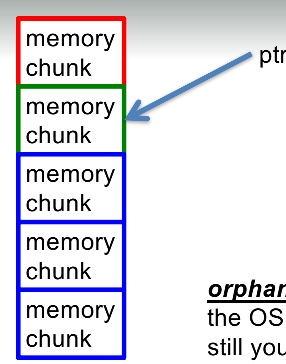


missing `delete [] ptr`

green, you are using
red, orphan
blue, free

orphan: you cannot use anymore but the OS doesn't know that, thinks it is still yours.

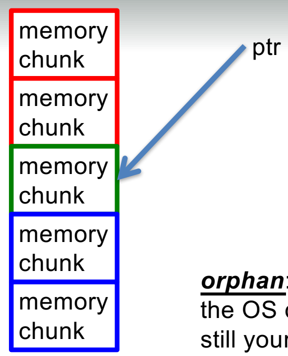
for (int i=0; i<10; ++i) ptr = new long[10];



green, you are using
red, orphan
blue, free

orphan: you cannot use anymore but the OS doesn't know that, thinks it is still yours.

for (int i=0; i<10; ++i) ptr = new long[10];



green, you are using
red, orphan
blue, free

orphan: you cannot use anymore but the OS doesn't know that, thinks it is still yours.

morale of the story

It is on you to free/delete memory that you have acquired

- there are consequences to this, and leaking memory is a problem.

Easiest way to avoid this:

Use the STL containers

- avoid the issue

scope, what's wrong with this fn?

```
long* fn1(size_t sz, long start,
          long inc ){
  auto ptr =  new long[sz];
  ptr[0]=start;
  for(int i=1; i<sz; i++)
    ptr[i]=ptr[i-1]+inc;
  return ptr;
}
```


who owns ptr?

A ptr in the function points to memory allocated in the function.

The ptr is deleted at the end of the function

- its address is returned to the caller

What happens to the memory?

local variables are deleted, but not memory

When the function returns, the ptr goes out of scope but the memory it points to does not:

- it still has to be deleted. It will leak otherwise.
- given the way this is set up, the calling program will have to delete it.