CIS 22C

Data Abstraction and Data Structures

Ron Kleinman

M1: Getting Started

Canvas Reading Assignment:

- Welcome Announcement, Syllabus, Greensheet

Intro Module: Logistics

- Goals of the course & some FAQs
- Student survey

Intro Module: Technical

- Importance of encapsulation
- C++ Language Review: Arrays & Pointers

CIS 22C Overarching Goal

'*Data Abstraction and Data Structures*' is designed to provide you with a solid understanding of the various ways one object type can collect another … and the ability to compare their relative strengths and weaknesses.

The journey begins

What exactly are classes & objects?

Class is a "Thing"

Food

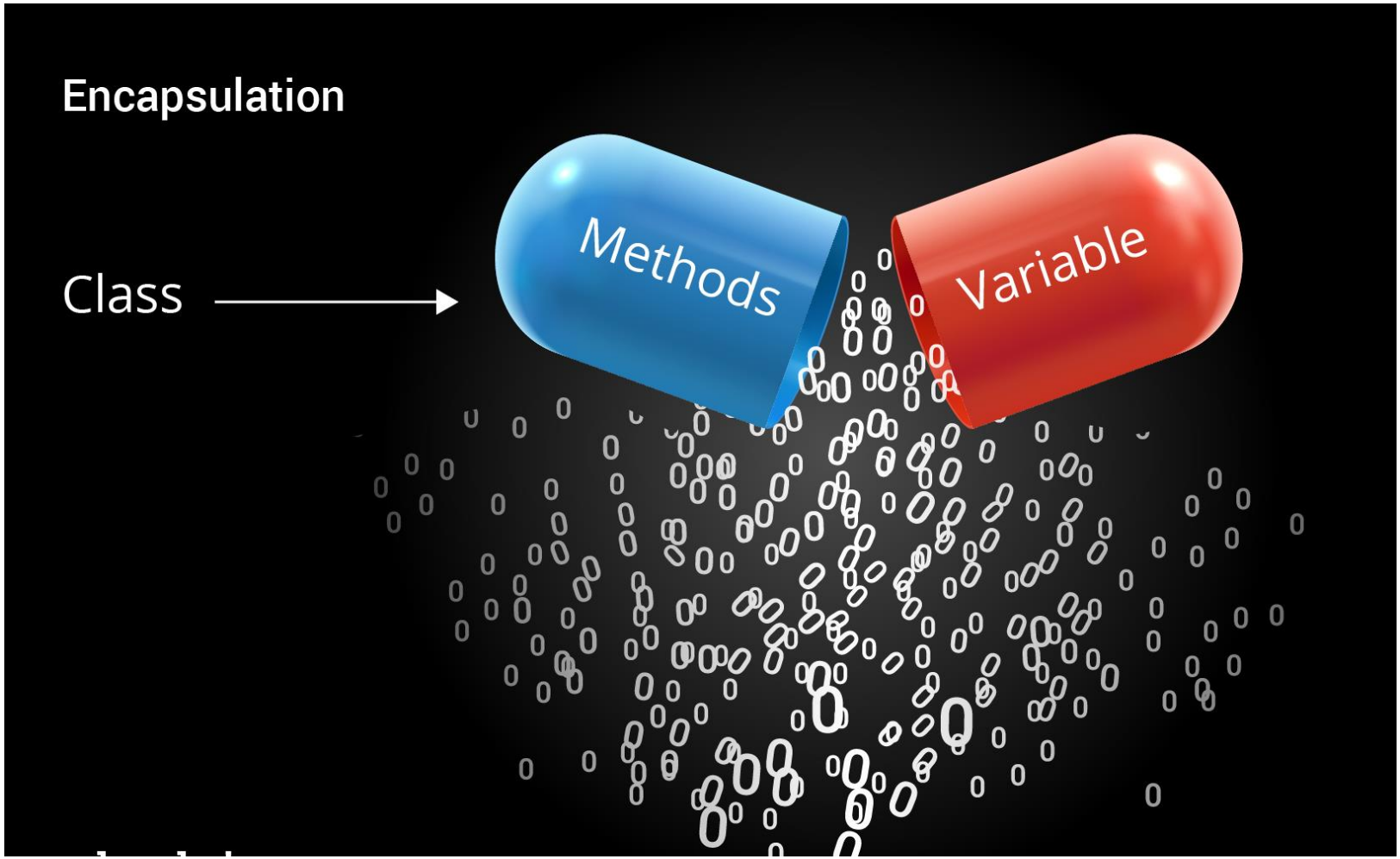
Body Organs

Gems

Object is a …

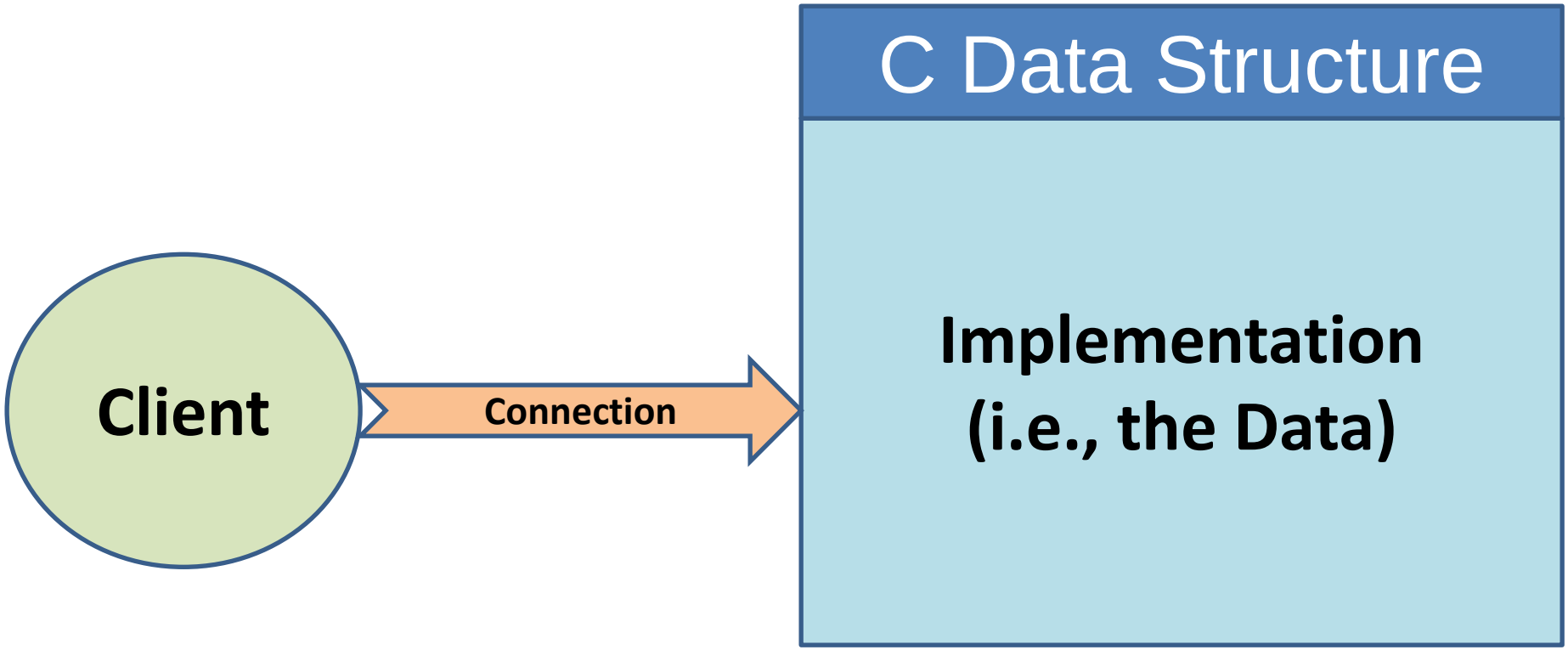
Thing "Instance"

What makes an object an object?



Objects

Client → C Data: Structure



C Structure internals visible

```
struct teacher
{
   String name;
   int id;
   Section *s[10];  // Array of Section ptrs
   String deptName;
} Teacher;
```

```
struct department
{
     String name;
     String dean;
     RoomList rooms;
     String description
} Department;
```

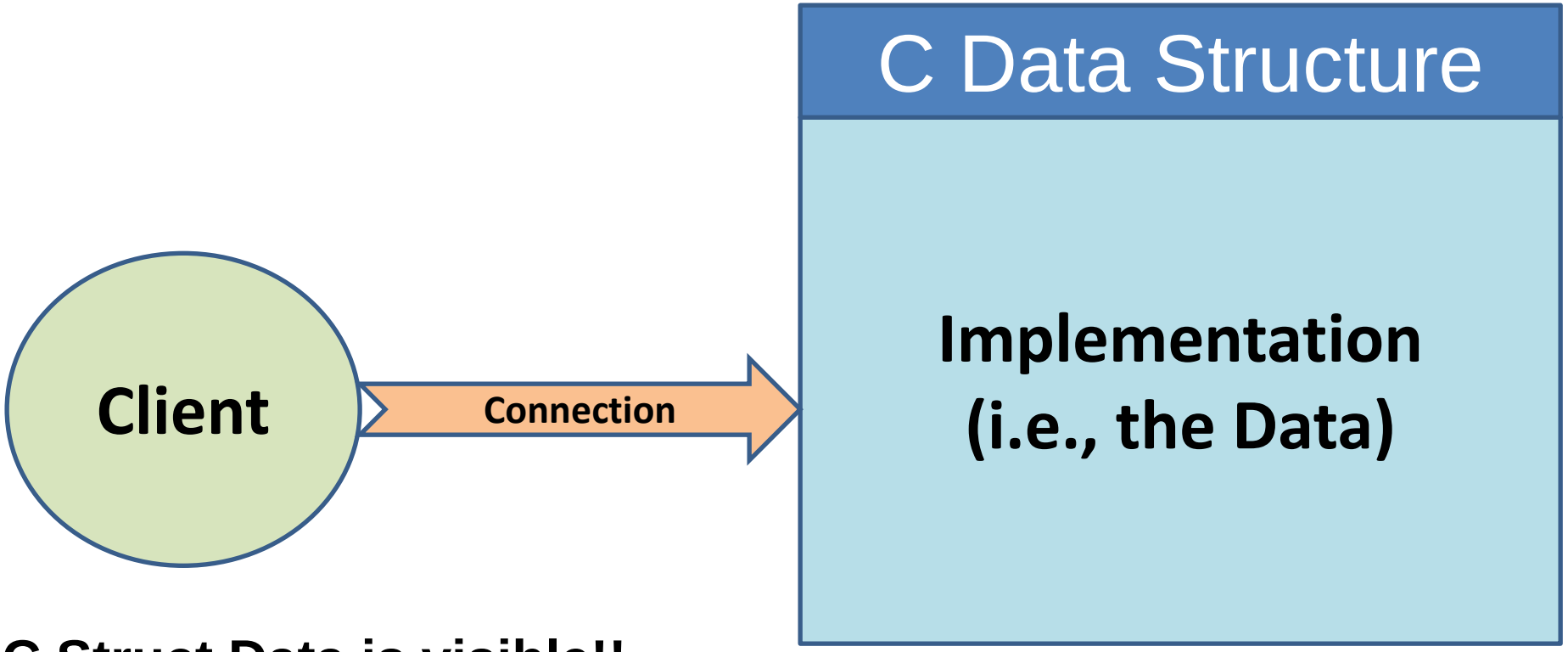
Client: Teacher t;
printf (“Department Name = %s\n“, **t.deptName**);

Internal Teacher Change: *String deptName* → *Department *dept*

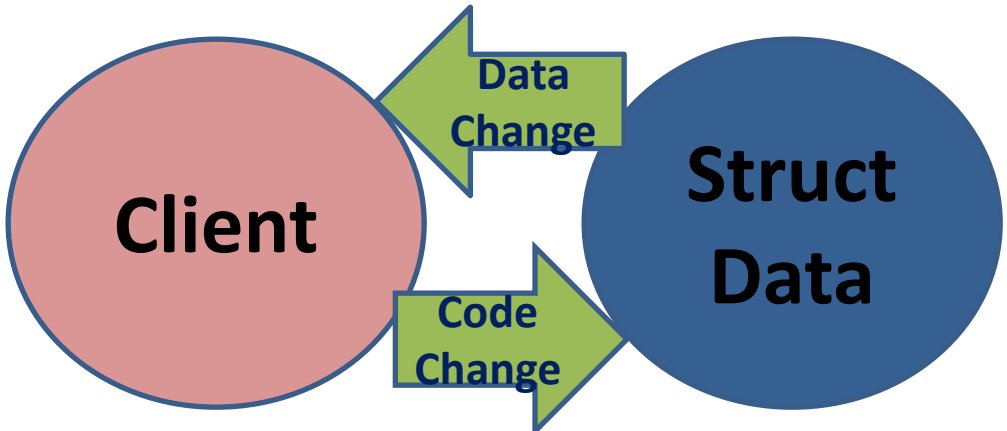
Client: Teacher t;
printf (“Department Name = %s\n”, **t->dept.name**);

→ Change in internal structure breaks all clients!

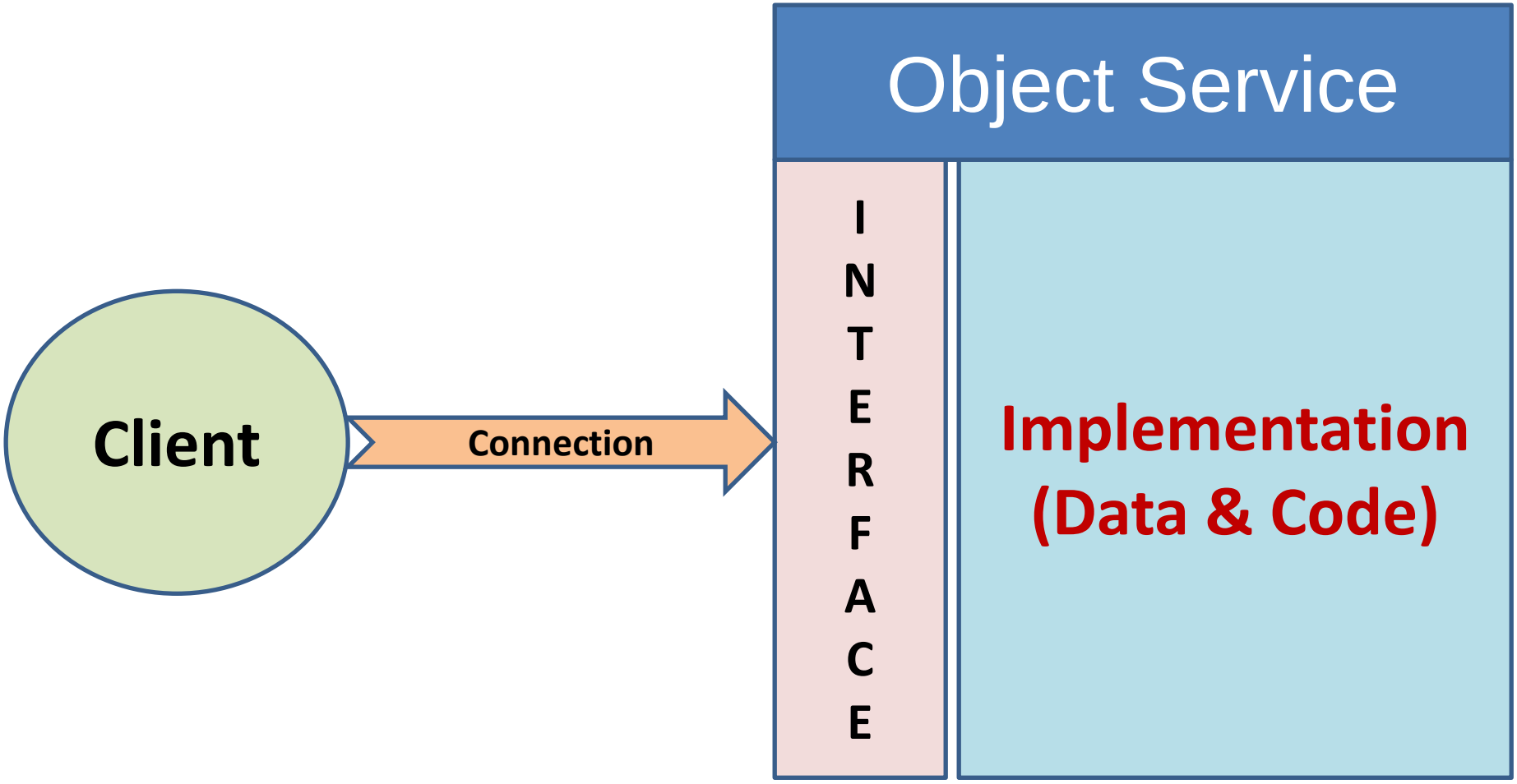
Clients and C Data



C Struct Data is visible!!



Clients and C++ Data: Objects!!



➔Interface (public methods) shield clients from internal data

➔Implementation can be developed independently of clients

➔Implementation can be refactored with no impact to clients

C++ Object Encapsulation

```
class Teacher
{
private:
   String name;
   int id;
   Section *s[10];  // Array of Section Ptrs
   String deptName;
public:
   String getDeptName ();
};
```

```
class Department
{
private:
     String name;
     String dean;
     RoomList rooms;
     String description
public:
    String getName();
};
```

Client:

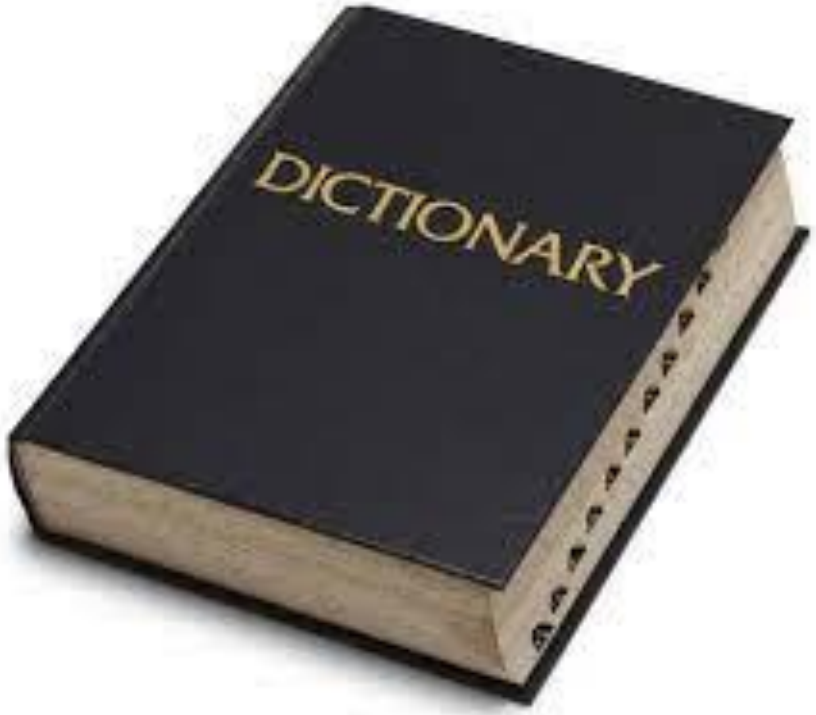
```
Teacher t;
 cout << “Department Name = “ << t.getDeptName () << endl;
```

Change internal Teacher variable *String deptName* to *Department *dept*

➔ internal code for getDeptName() is all that changes

➔ NO change to any clients of that object!

Defining the words



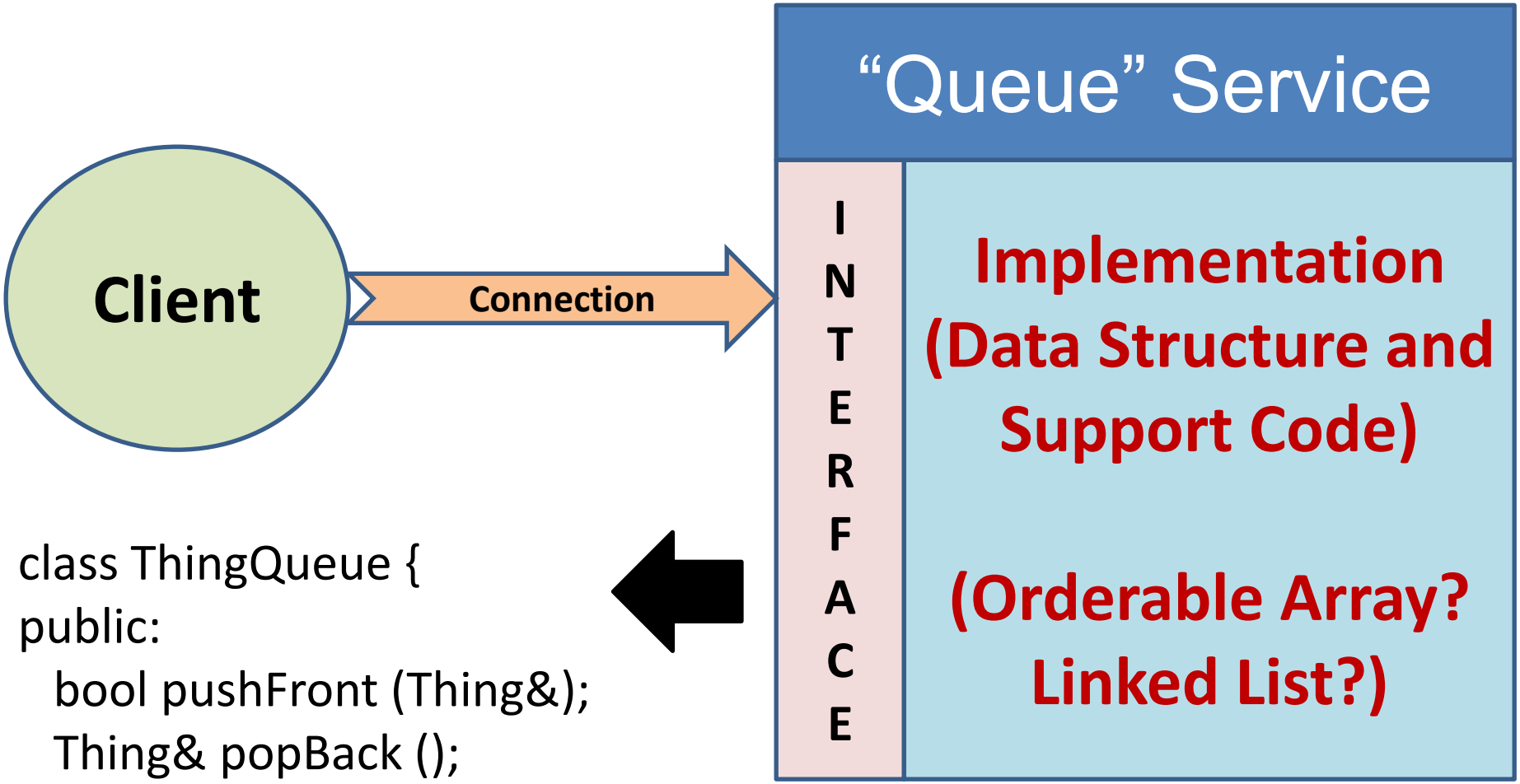
"Data Abstraction and Data Structures"

(Meanings? Differences?)

Course Vocabulary

- **Collection:** Group of multiple Elements
  - Contained Elements are "homogeneous"
- **Data Abstraction:** Collection Interface
  - Ex: Stack (LIFO), Queue (FIFO), …
- **Data Structure:** Collection Implementation
  - Ex: Orderable Array, Linked List, …
- **"Purpose": An "operation" on data in a Collection**
  - Ex: Order (sort), Merge, randomize, detect duplicates, ….
- **Algorithm:** Step by step strategy for implementing that purpose
  - Ex: Bubble Sort, Binary Search, …

Data Abstraction hides underlying Data Structure!




```
class ThingQueue {
public:
   bool pushFront (Thing&);
   Thing& popBack ();

   Thing& peek();
   int getSize();
 }
```

➔ Any Data Abstraction might be implemented using any of several Data Structures!!

A brief overview of the C++ Toolset

Arrays and Ptrs: The starter Kit

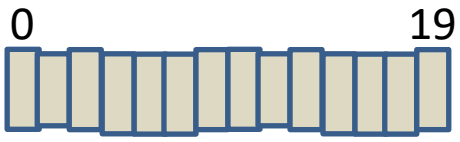
- Arrays vs. Ptrs
  - How implemented in C++
  - Functionality & Limitations

- Element Variations
  - Arrays of Basic Data Types
  - Arrays of Objects
  - “2nd level” { Pointer / Array } combinations

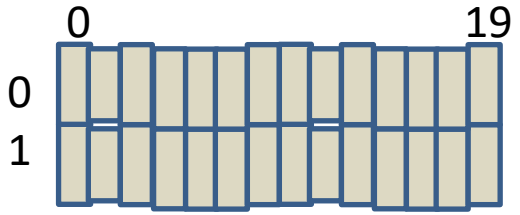
Basic Array Functionality

- **Homogeneous Collection**
  - Elements all the same object type
- **Fixed Size** (initial space cannot be extended)
- **No boundary checking** (overflows happen)
- **Element Retrieval Key** is unsigned integer position
  - All ordering is “external”
- **Single or multi-dimensional**

Thing a[20];



Thing b[2][20];



Arrays and Pointers: Questions

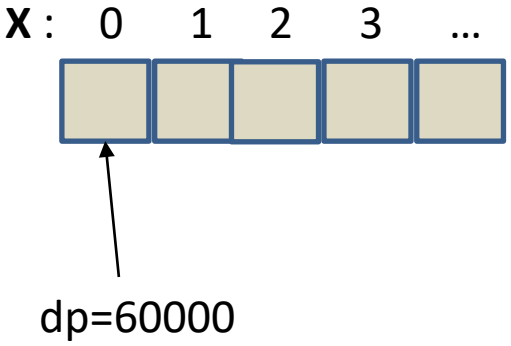
(Assume double X[10]; double *dp; int *ip;)

- **dp = &X[0];** // dp contains byte address of X[0]

  dp++; // How can dp now point to X[1] (8 bytes away)??

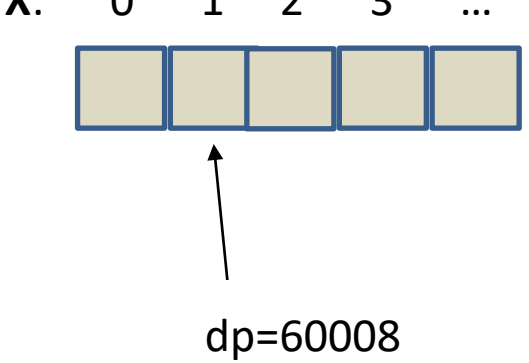
- **Why can't any pointer type be equated to address x?**

  ip = X; // Fails(??) **Isn't every ptr value an address??**



X IS 60000
(Memory Address
where array begins)

dp++;



Arrays and Pointers: Answers

- **How can incrementing (adding 1) to a Thing pointer which points to element X in a Thing array, result in its pointing to element X+1, no matter what the size of the Thing actually is?**

  Arithmetic operators in C/C++ do different things when applied to pointers which point to different element types (intrinsic pointer arithmetic)

- **Why can't any pointer type be equated to X?**

  Intrinsic C / C++ pointer arithmetic.

Intrinsic C / C++ Pointer Arithmetic

Supported operations (Thing *tp1, Thing *tp2; int x)

- **Subtraction between 2 Thing pointers** (ex: tp1 - tp2)
  - Value returned is *(tp1 - tp2) / sizeof (Thing)* … or the difference in the # of Things between the 2 Thing pointers (and **not** the difference in the # of bytes).

- **Adding / subtracting an integer from a pointer (ex: tp1+x)**
  - Value returned is *tp1 + (x * sizeof (Thing))* … or the byte value of the Thing pointer moved forward x things (and **not** the value moved forward x bytes).

Now that we know what an array is ... and isn't

What are its limitations?

What sorts of "things" can it collect?

How are arrays optimally used in a real-world application?

Array Limitations

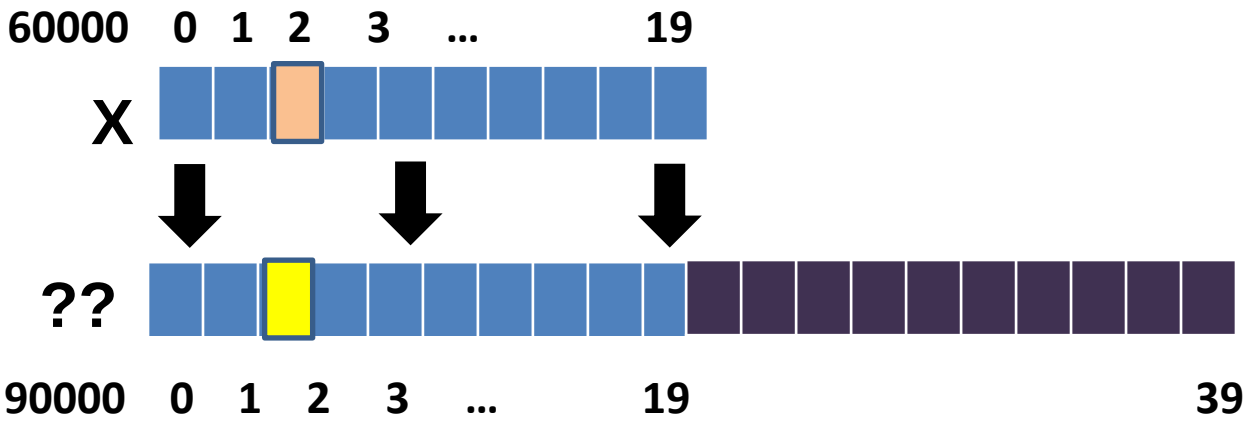
- **No boundary checks on dynamically calculated index**
  - int y = 5; double x[10]; x[4*y] = 3.14159; // Kerblooey!!
- **No auto-resizing when array hits size limit**
  - Allocate memory for a larger new array
  - Copy all elements of old array to start of new array
  - Free old array
- **Deleting the x[j]th element in Array can be cumbersome**
  - Move every element from j to n-1 forward one
- **Inserting an x[j]th element can be even worse**
  - Check if resizing needed and if so, do it manually (see above)
  - Push every element from j to n-1 backward one
  - Insert value in x[j]

Resizing an Array when max limit reached

<Array resizing occurs!>

Each Array Element is:

1. Copied to a new location
2. Destructed



Simple Array Element types

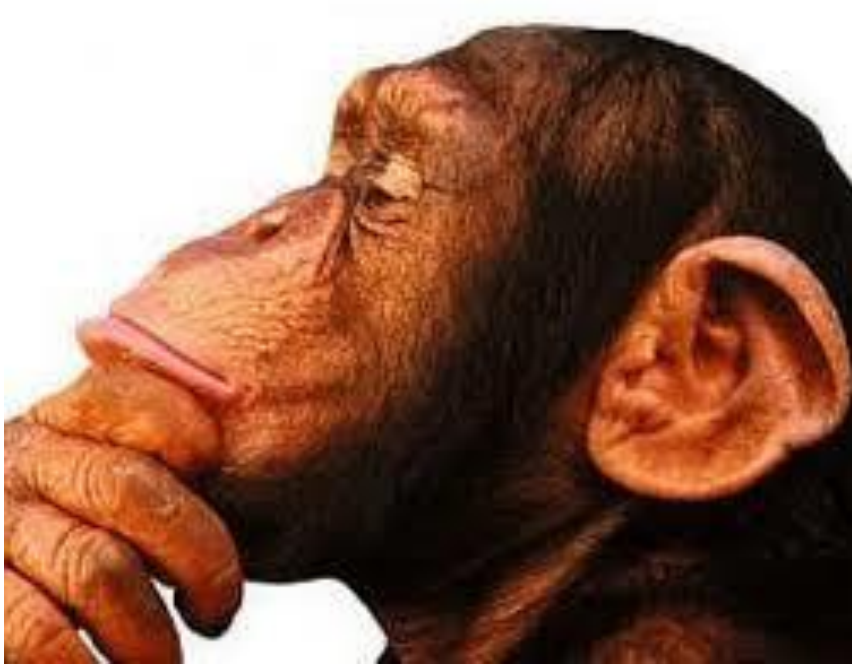
- **Intrinsic Basic (**int, float, char, …)
  - Sortable: Comparison operators (==, >, <, !-,…) defined
  - “Copy contents” provided via equate “=“ operator

- **Structs**
  - Not sortable (no struct “>” operator support in C++)
  - Shallow equate (value duplicated, not what was ptd to)

A brief C++ Tutorial before pushing on

- Given that **Thing is an object <u>type</u>**

  Thing t; // t is a Thing object. What is:



Level 1

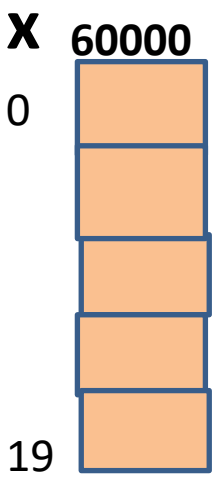
- **Thing* tp;**
- **Thing ta[100];**

Level 2

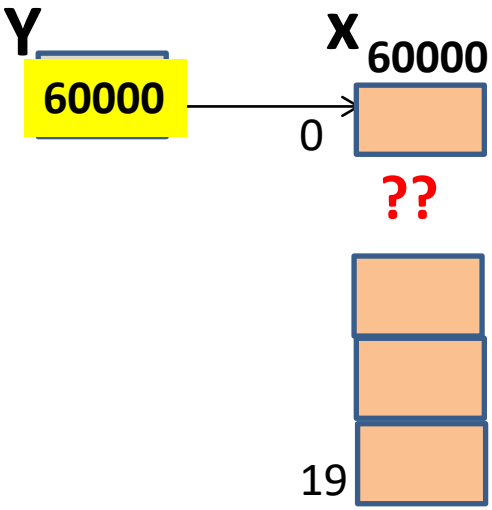
- **Thing* *tpp;**
- **Thing* tap[100];**
- **Thing (*tpa)[100];**
- **Thing taa[20][100];**

Array Element: Object vs. Array vs. Pointer

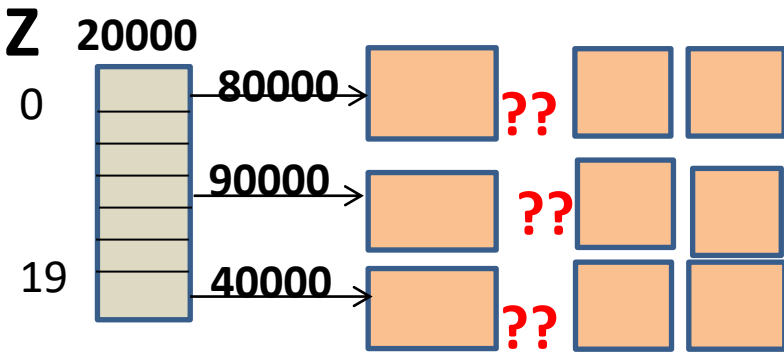
Thing X[20];

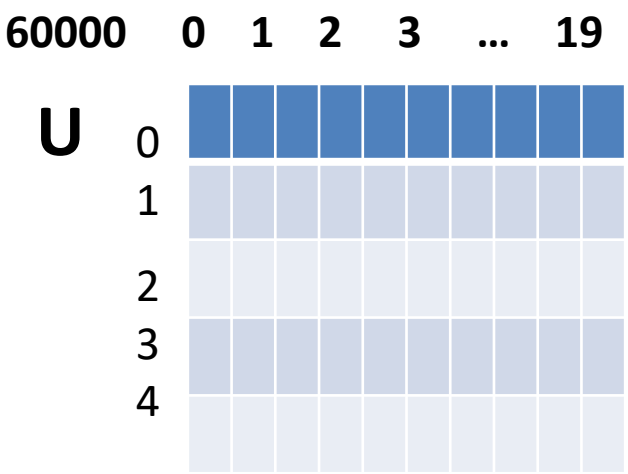


Thing (*Y)[20];

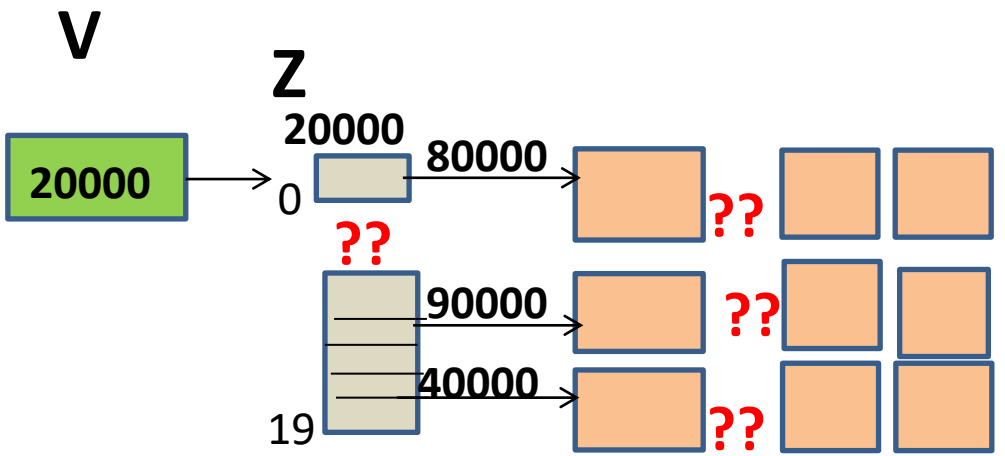


Thing *Z[20];





Thing U[5][20]



Thing ** V

A brief C++ Tutorial before pushing on

- Given that **Thing is an object <u>type</u>**

  Thing t; // t is a Thing object

Level 1

- **Thing* tp;** // tp points to a Thing
- **Thing ta[100];** // ta is an array of 100 Things

Level 2

- **Thing* *tpp;** // tpp points to a Thing pointer
- **Thing* tap[100];** // tap is an array of 100 Thing ptrs
- **Thing (*tpa)[100];** // tpa points to a 100 Thing Array
- **Thing taa[20][100];** // taa is an array of 20 "*100 Thing*" Arrays

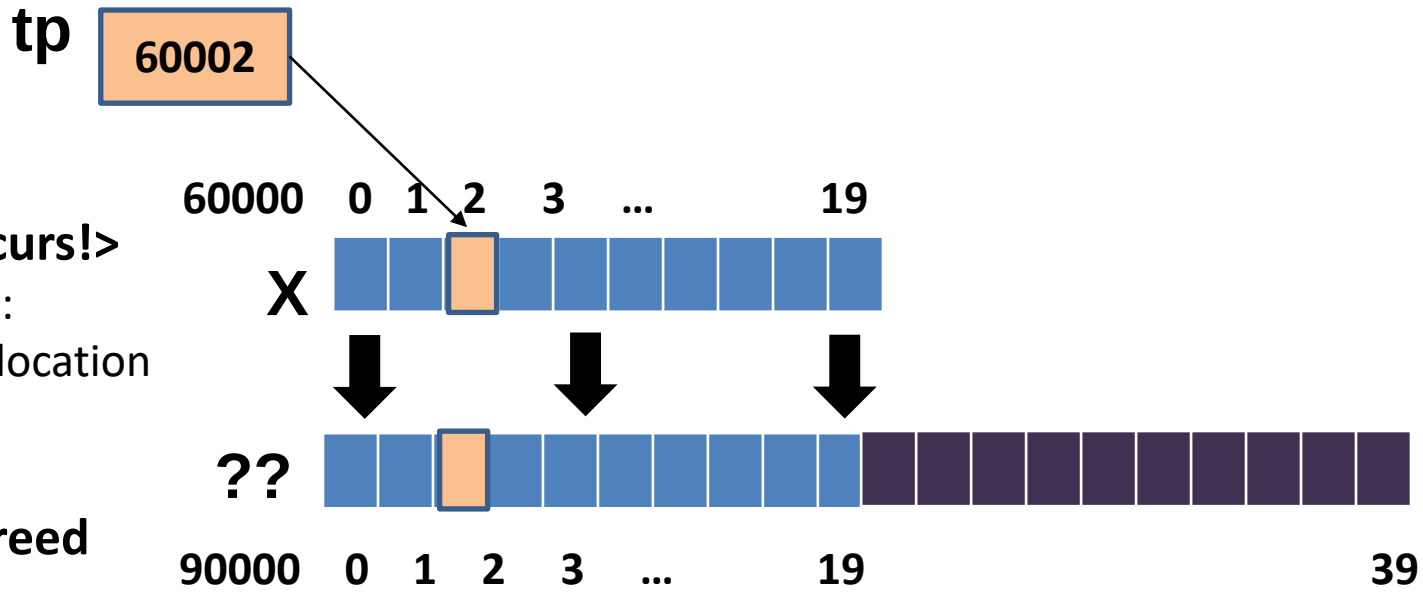
Array Element type: Object vs. Object Ptr

Element types: Objects vs. Object Ptrs

- **Any Object Element in an Array:**
  - Default Constructor
    - Invoked at Array Creation (every element)
  - Copy Constructor
    - Invoked on new element during Array resizing
  - Destructor
    - Invoked on old element during Array resizing
- **Any Object Pointer Element in an Array:**
  - No constructors or destructors invoked by Array ops
  - “Shallow” copies (ptrs duplicated, not what is ptd to)

Resizing an object Array (max limit reached)

Thing *tp = &X[2];



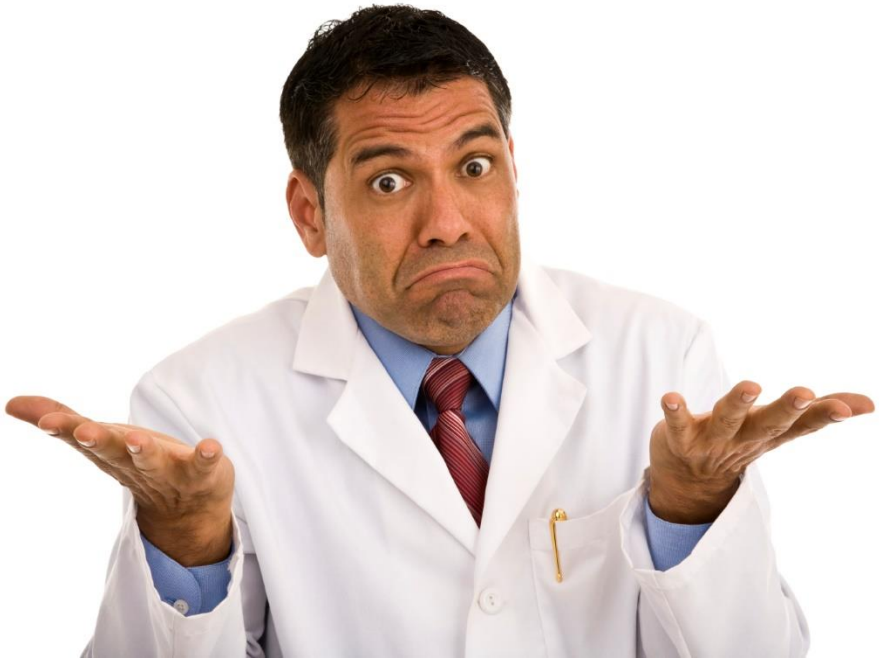
<Array resizing occurs!>
Object tp points to is:
1. Copied to a new location
2. Destructed

tp now points to freed memory!!!

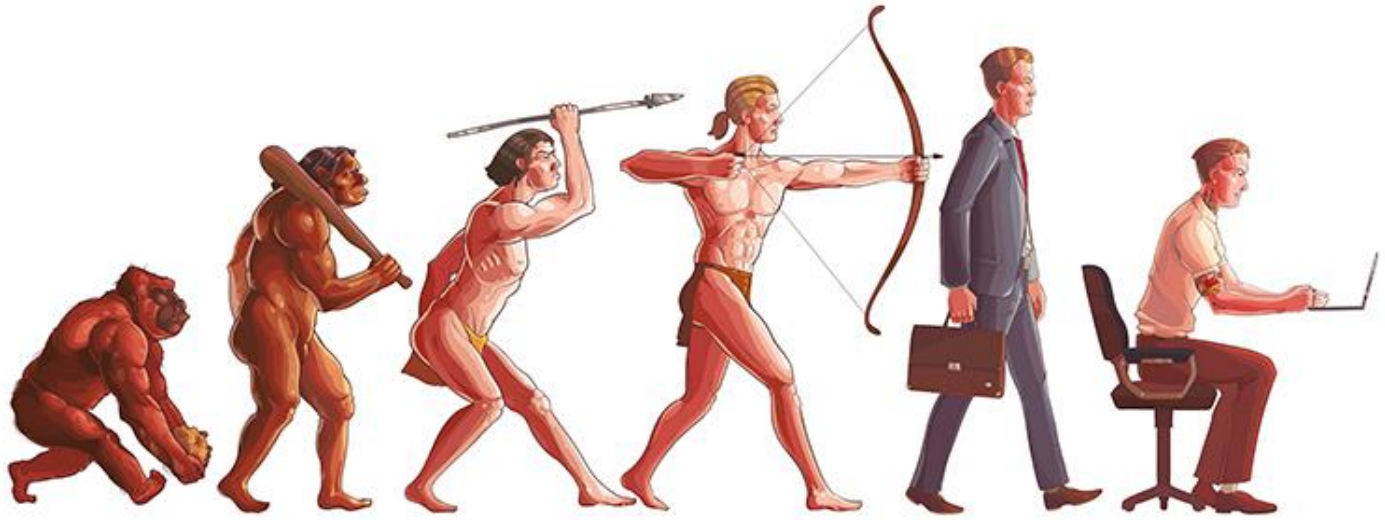
Any array of objects <u>whose elements are pointed to by external pointers</u> (ie. Pointers not under the control of the array owner), cannot be resized!!

➔ Enough space must be allocated so that resizing is never necessary.

But when are the contents of an object Array ever pointed to externally by pointers not under the control of the Array owner?



How did “objects” get here?



Condensed 30-year history of the evolution of programming languages

You can't build a house without a Foundation

Evolution of programming languages was driven by:

#1 The desire to allow a direct mapping between a data abstraction (like "Teacher") and programming language data constructs (int, float, char). **<struct>**

#2 The desire to raise a wall around the code & data details of one "module" to hide it from the other modules. **<class>**

From the ridiculous …

"Bill Gates" Basic does Sales Tax in 1970

```
100 Rem Call the compute sales tax subroutine
105 Rem a7 must have the total earnings, a8 the state tax %, a9 will be clobbered
110 gosub 6000
120 Print "Sales Tax is"; a9;
…
6000 Rem Compute Sales Tax
6004 Rem a7 has the total earnings, a8 has the state tax %
6008 a9 = a8 * a7;
6012 return
```

All subroutines must be in same program file

Subroutines have no name

Subroutines take and return no arguments

Subroutines have no local variables (a7, a8, a9 are GLOBAL)

Language	**Code Modules**	**Variable Scope / Visibility**	**Variable "Meaning"**	**Operations**
	Program	Global	Fixed Set (int, float, string)	Fixed Set (+-*/)
Fortran	Subroutines	**Global** + local to Sub	Fixed Set	Fixed Set
C	Functions	Prog Global File Global Funct local Loop Local	Fixed Set + **Data Structure**	Fixed Set and =, != operate on struct
C++	Objects	All C varieties	All C varieties + Class adds (Complex, …)	Any operator definable for class (+,-, …)

To the sublime(?) …

C++ object does Sales Tax

```
double myincome;      // Previously set to annual income
TaxCalulator taxCalc; // Create a tax calculator object
enum usState ms = CA      // Set "US State" enum to California
double salestax = taxCalc.getSalesTax (myincome, ms); // Magic!!

// TaxCalc:
// No object variables or logic visible.  "Specs" are public methods
// Could be independently written.  Could be … reused!!
class TaxCalculator   {
  public:
// Return State Sales Tax due from total income and State
     double getSalesTax (double totIncome, enum usState state) {};
```