C++ Object-Oriented Prog.

Unit 4: Objects and Lists

CHAPTER 15: DYNAMIC DATA AND LINKED LISTS

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Overview of Using Linked List

Chapter 15 Topics

- Using the Address-Of Operator &
- Declaring and Using Pointer Variables
- Using the Indirection (Dereference) Operator *
- The NULL Pointer
- Using C++ Operators new and delete
- Meaning of an Inaccessible Object
- Meaning of a Dangling Pointer
- Use of a Class Destructor
- Shallow Copy vs. Deep Copy of Class Objects
- Use of a Copy Constructor

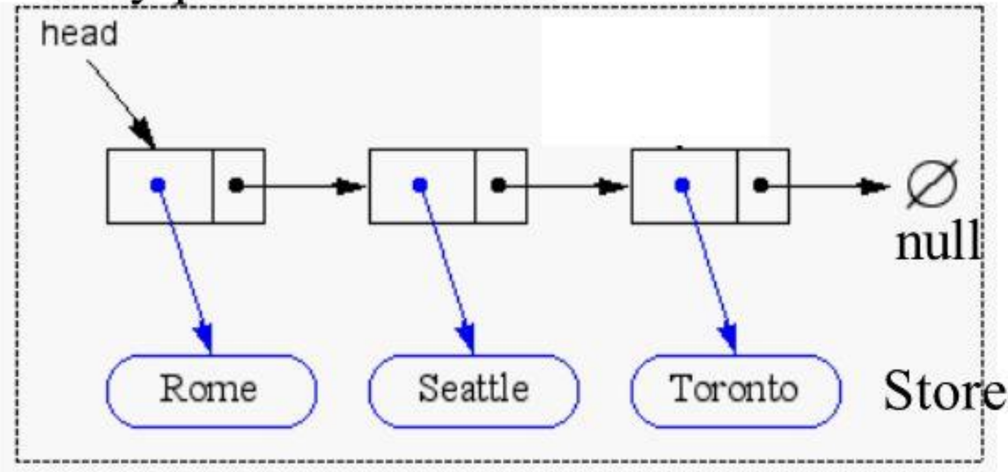
Linked lists

Arrays are one way to implement Stacks, queues, etc.

Linked Lists are another -- extremely flexible and general idea!

Linked list = "Node" objects connected
in a "chain" by links (object references)

Special "entry point" reference



Stored values (eg, Strings)

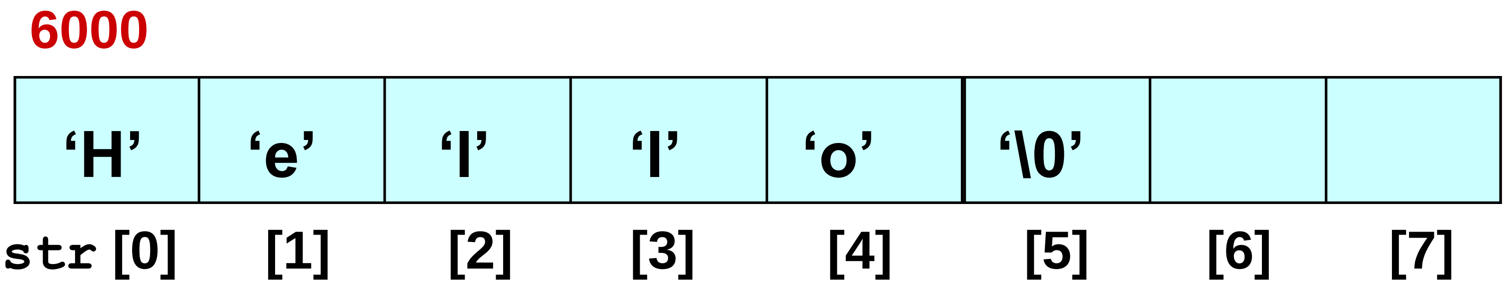
Boxes are "Node" objects

(*<u>not</u>* 'built in' -- you must define/manage them!)

Recall that . . .

```
char  str [ 8 ];
```

- `str` is the base address of the array.
- We say `str` is a pointer because its value is an address.
- It is a pointer constant because the value of `str` itself cannot be changed by assignment.
- It "points" to the memory location of a `char`.



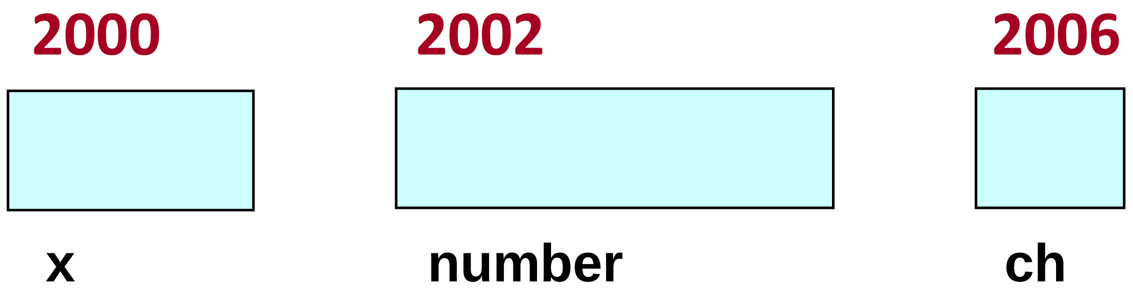
6000

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’		
`str` [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Addresses in Memory

When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable

```
int      x;
float    number;
char     ch;
```



Obtaining Memory Addresses

- the address of a non-array variable can be obtained by using the address-of operator &

```
int       x;
float     number;
char      ch;
cout << “Address of x is ” << &x << endl;
cout << “Address of number is ” << &number << endl;
cout << “Address of ch is ” << &ch << endl;
```

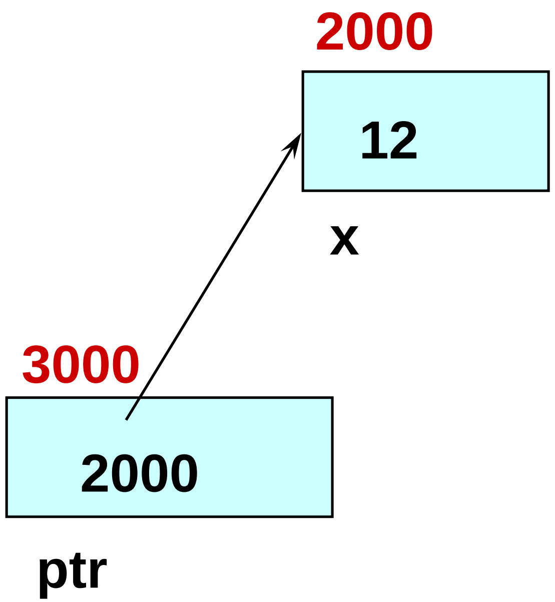
What is a pointer variable?

- A pointer variable is a variable whose value is the address of a location in memory
- To declare a pointer variable, you specify the type of value that the pointer will point to, for example

```
int*   ptr; // ptr will hold the address of an int
char*  q;   // q will hold the address of a char
```

Using a Pointer Variable

```
int  x;
x = 12;


int*  ptr;
ptr = &x;
```



2000

12

x

3000

2000

ptr

NOTE: Because ptr holds the address of x,

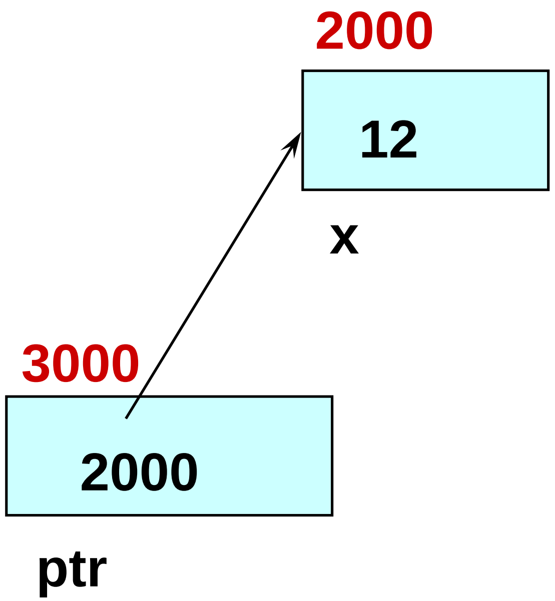
we say that ptr “points to” x

LECTURE 2

Dereference *

Unary operator * is the indirection (deference) operator

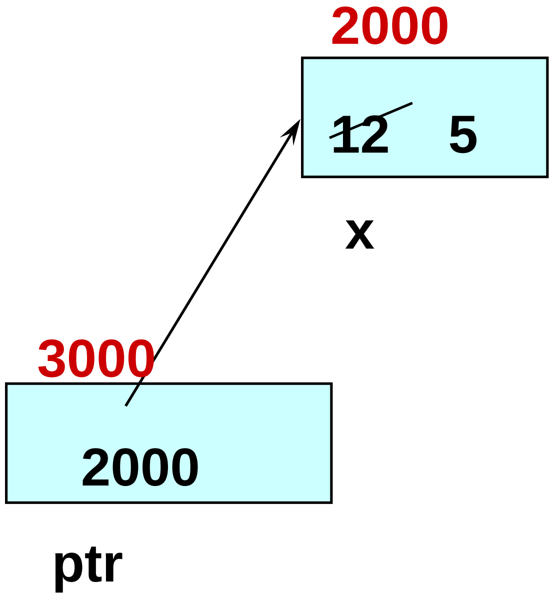
```
int   x;
x = 12;
int*  ptr;
ptr = &x;
cout  <<  *ptr;
```



NOTE: The value pointed to by ptr is denoted by *ptr

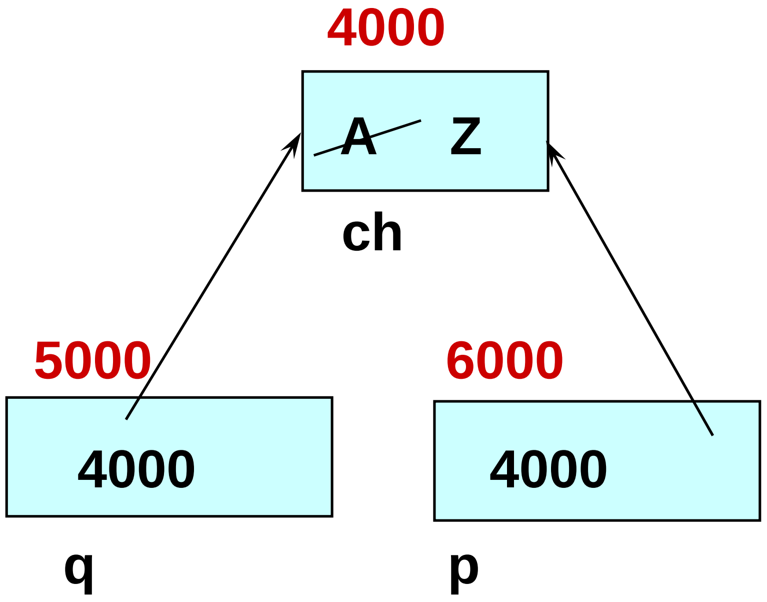
Using the Dereference Operator

```
int  x;
x = 12;


int*  ptr;
ptr = &x;

*ptr = 5; // Changes the value
          //  at address ptr to 5
```



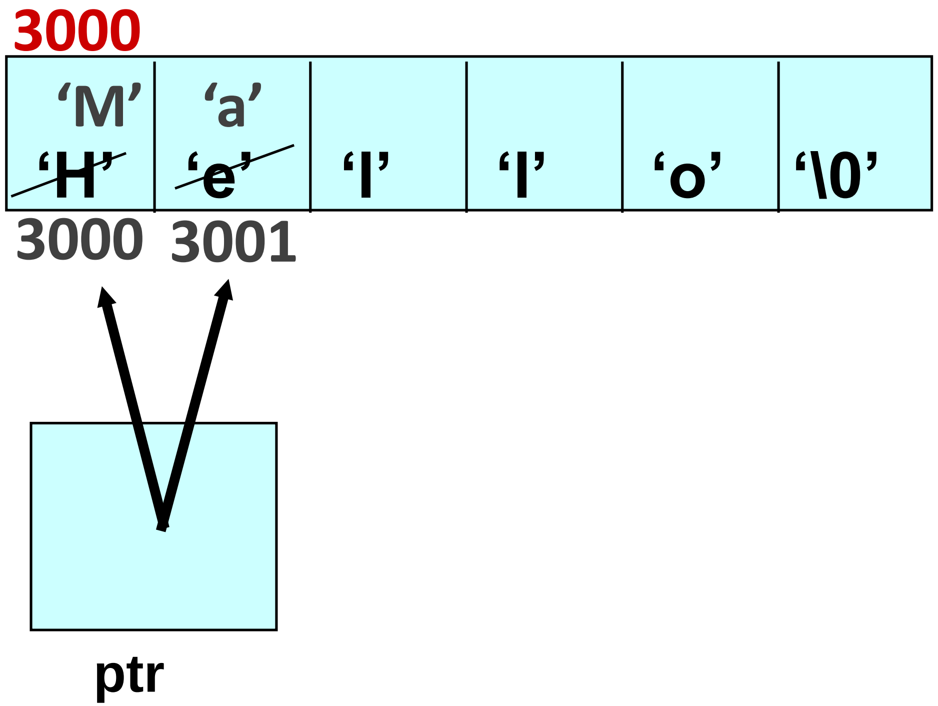
Another Example

```
char   ch;
ch =   'A';
char*   q;
q   = &ch;
*q = 'Z';
char*   p;
p = q;  // The rhs has value 4000
        // Now p and q both point to ch
```



Using a Pointer to Access the Elements of a String

```
char msg[ ]="Hello";
char*  ptr;
ptr   =   msg; // Recall that msg ==
              //   &msg[ 0 ]
*ptr   =  'M';
 ptr++;       // Increments the address
*ptr = 'a';   // in ptr
```



```
int StringLength (/* in */ const char str[] )
// Precondition: str  is a null-terminated string
// Postcondition:  Return value == length of str
//   (not counting ‘\0’)
{
    char* p;
    int count = 0;

    p = str;

    while (*p != ‘\0’)
    {
        count++;
        p++;
        // Increments the address p by sizeof char
    }

    return count;

}
```

Indexing a pointer

These two parameters are the same

```
    char str[]          char* str
```

Indexing a pointer is allowed (whether the pointer points to an array or not)

Indexing is valid for any pointer expression

Character array char a[] is different from string c

1. C’s character array is still a valid data type in C++.

2. Each character is accessed as **a[i]**

3. C’s character array size is calculated by sizeof(a)/sizeof(char)

4. C++’s string is iterable. C++ string has many function similar to vector. C++ stirng is one less than sizeof(a)/sizeof(char) because of ‘\0’

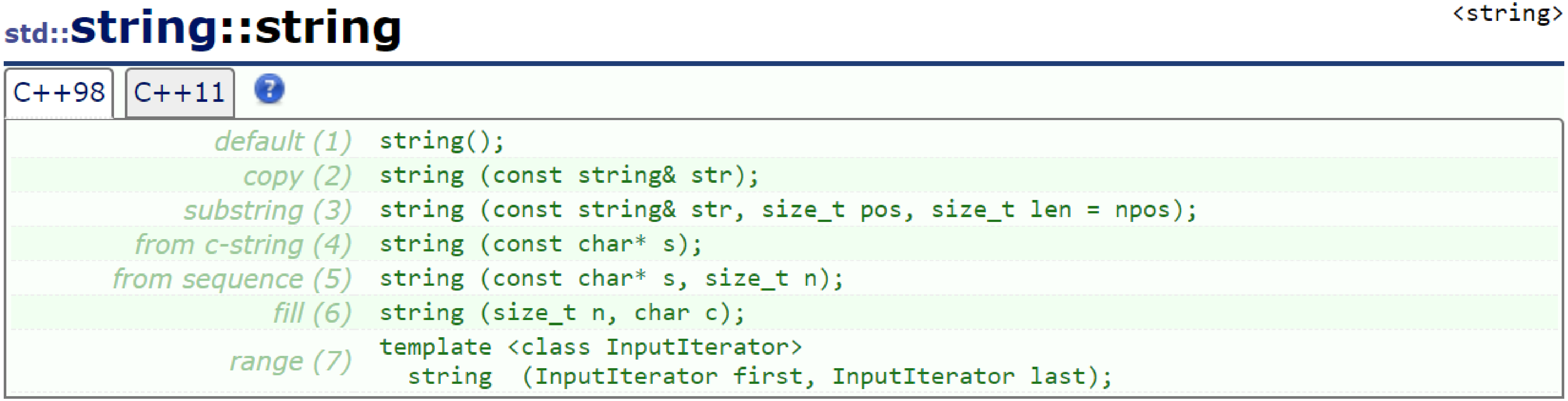
5. c[i], c.at(i), *it are used to access the character indexed at i.

6. C++ string has length() method.

Demo Program: string0.cpp

Go Notepad++!!!

Using string constructor to convert a char array into string



std::string::string

<string>

default (1)	`string();`
copy (2)	`string (const string& str);`
substring (3)	`string (const string& str, size_t pos, size_t len = npos);`
from c-string (4)	`string (const char* s);`
from sequence (5)	`string (const char* s, size_t n);`
fill (6)	`string (size_t n, char c);`

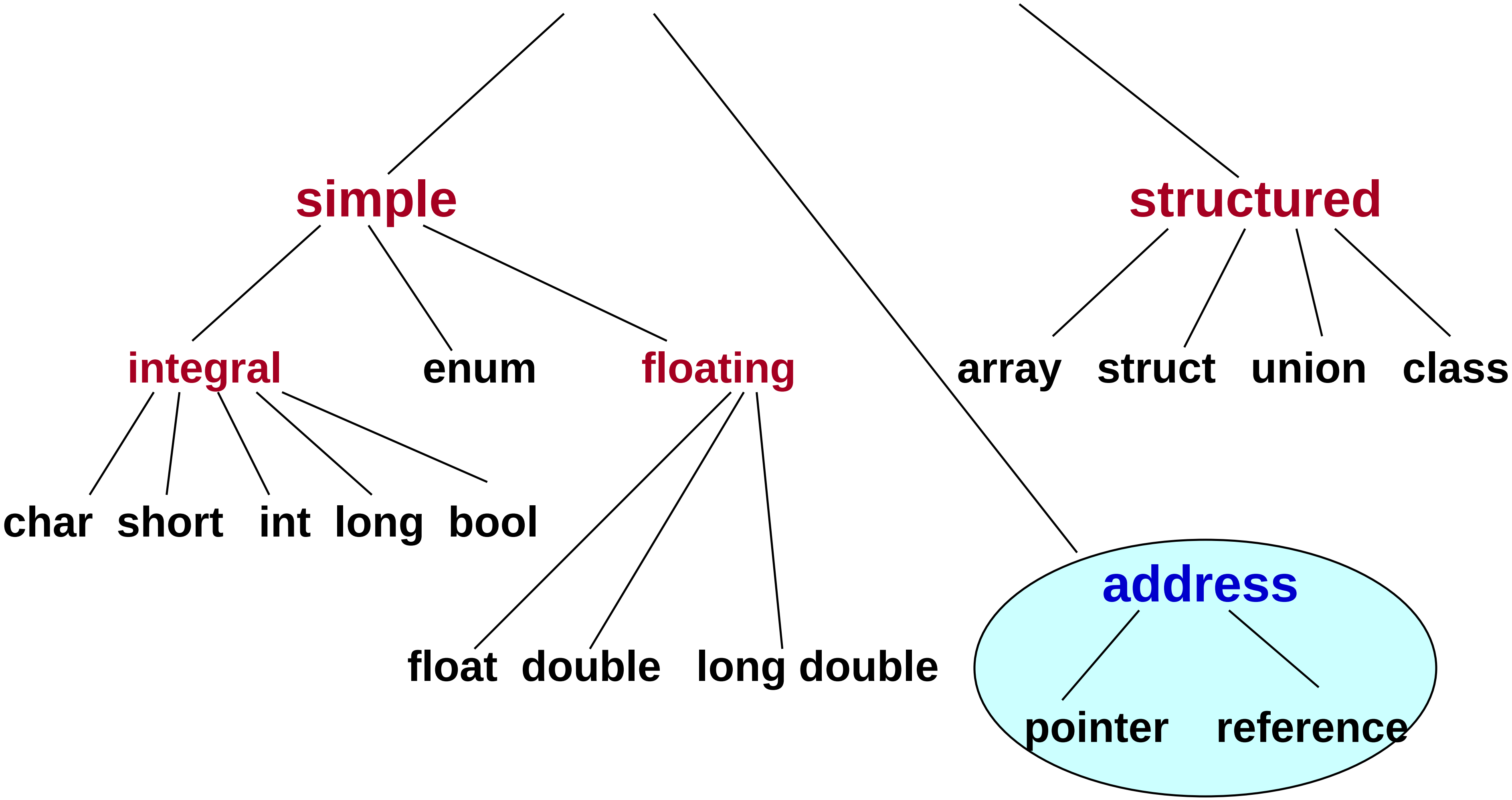
range (7)	`template <class InputIterator>`
	`  string  (InputIterator first, InputIterator last);`

It is not hard to convert a C++ back to C character array, since we know the length of a string and we know how to traverse through a C++ string.

LECTURE 3

new Operator, * and & static, auto and dynamic data

C++ Data Types


simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
address
float
double
long double
pointer
reference

Some C++ Pointer Operations

Precedence		
Higher	->	**Select member of class pointed to**
	Unary: ++ -- ! * & new delete	Increment, Decrement, NOT, Dereference, Address-of, Allocate, Deallocate
	Binary: + −	Add Subtract
	< <= > >=	Relational operators
	== !=	Tests for equality, inequality
Lower	=	Assignment

Operator `new` Syntax

```
new   DataType
```

```
new   DataType  [IntExpression]
```

- If memory is available in an area called the heap (or free store) new allocates space for the requested object or array and returns a pointer to (address of) the memory allocated
- Otherwise, program terminates with error message
- The dynamically allocated object exists until the delete operator destroys it

The NULL Pointer

- NULL is a pointer constant 0, defined in header file cstddef, that means that the pointer points to nothing
- It is an error to dereference a pointer whose value is NULL
- Such an error may cause your program to crash, or behave erratically

```
while (ptr != NULL)
  {
    . . . // Ok to use *ptr here
  }
```

3 Kinds of Program Data

Static data: memory allocation exists throughout execution of program

```
static long currentSeed;
```

Automatic data: automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function

Dynamic data: explicitly allocated and deallocated during program execution by C++ instructions written by programmer using operators `new` and `delete`

Allocation of Memory

STATIC ALLOCATION

Static allocation is the allocation of memory space at compile time

DYNAMIC ALLOCATION

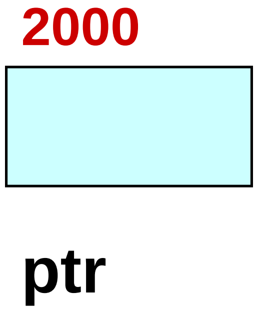
Dynamic allocation is the allocation of memory space at run time by using operator new

LECTURE 4

Dynamic Allocated Data

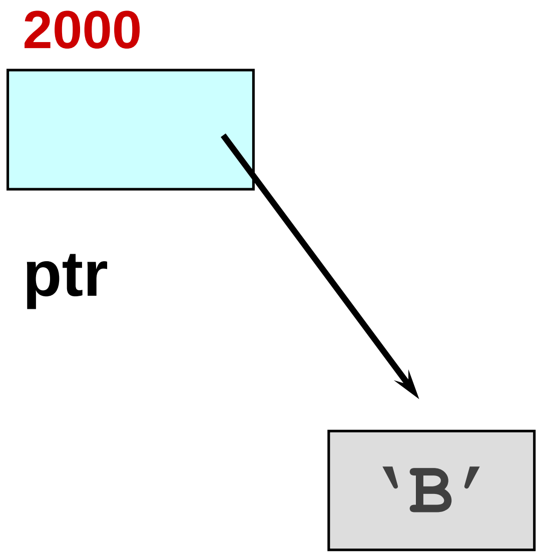
Dynamically Allocated Data

```
char*  ptr;


ptr = new char;
*ptr = ‘B’;
cout  <<  *ptr;
```



Dynamically Allocated Data

```
char*  ptr;
ptr = new char;

*ptr = 'B';

cout  <<  *ptr;
```

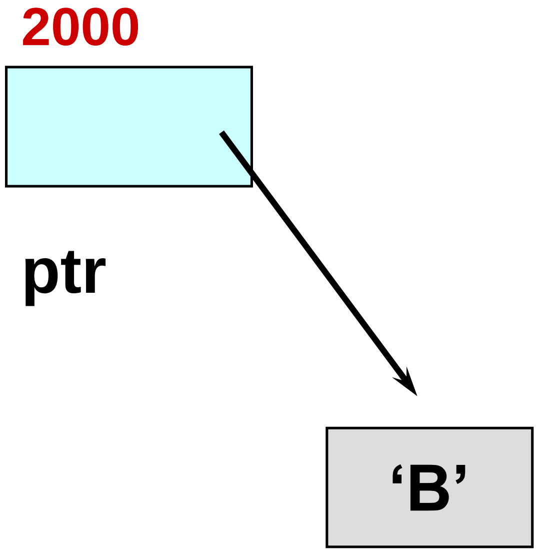


NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
char*  ptr;
ptr = new char;
*ptr = ‘B’;
```

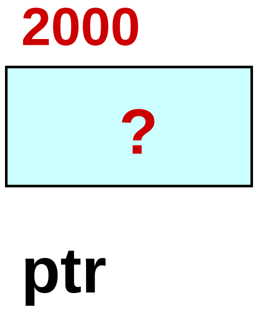
```
cout  <<  *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
char*  ptr;
ptr = new char;
*ptr = ‘B’;
cout  <<  *ptr;
delete  ptr;
```



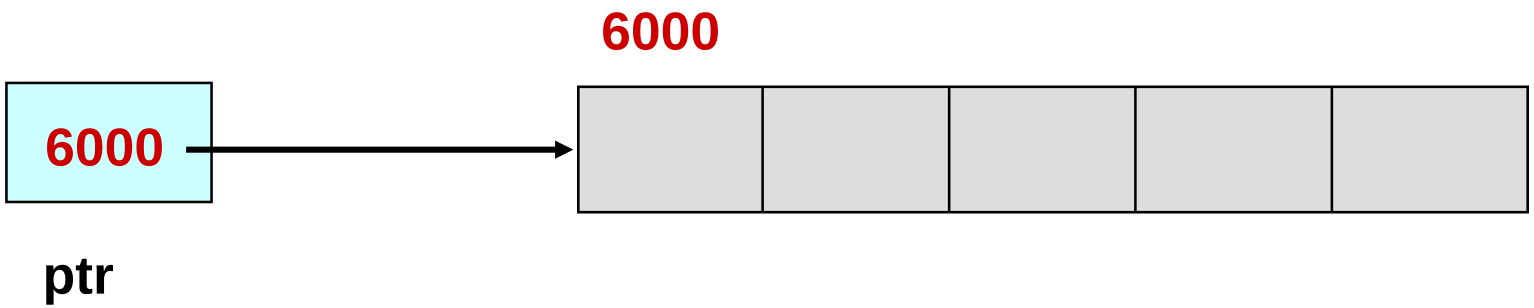
NOTE: delete deallocates the memory pointed to by ptr

Using Operator `delete`

- Operator delete returns memory to the free store, which was previously allocated at run-time by operator **new**
- The object or array currently pointed to by the pointer is deallocated, and the pointer is considered unassigned

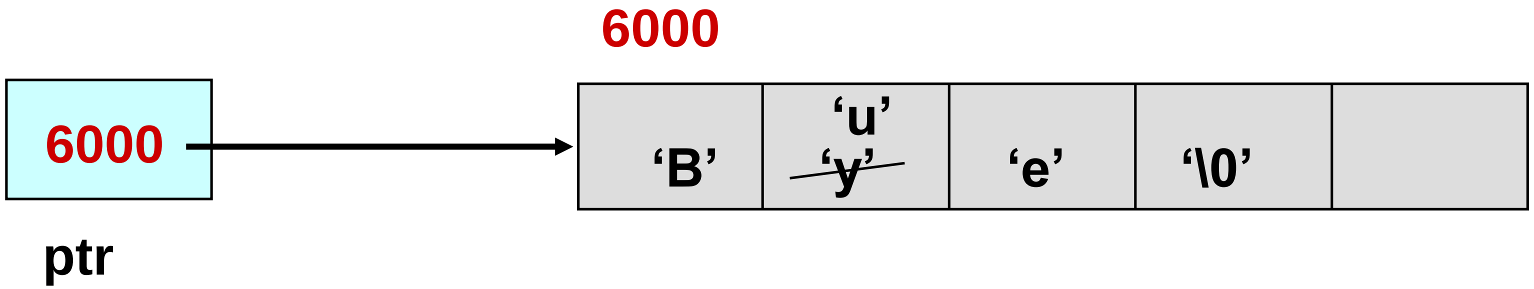
Dynamic Array Allocation

```
char *ptr;// ptr is a pointer variable that
          //  can hold the address of a char
ptr  =  new  char[ 5 ];
// Allocates memory for a 5-character array
//  dynamically at run time and stores the
//  base address into ptr
```



Dynamic Array Allocation

```
char  *ptr;

ptr  =  new  char[ 5 ];

strcpy(ptr, “Bye”);

ptr[ 1 ] = ‘u’;
// A pointer can be subscripted
cout  << ptr[ 2];
```



Operator `delete` Syntax

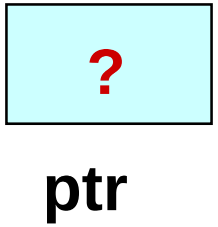
```
delete   Pointer
```

```
delete  [ ]   Pointer
```

- If the value of the pointer is NULL there is no effect
- Otherwise, the object or array currently pointed to by Pointer is deallocated, and the value of Pointer is undefined
- The memory is returned to the free store
- Square brackets are used with delete to deallocate a dynamically allocated array

Dynamic Array Deallocation

```
char *ptr;

ptr = new char[ 5 ];

strcpy(ptr, “Bye”);
ptr[ 1 ] = ‘u’;
delete [] ptr;
// Deallocates array pointed to by ptr
// ptr itself is not deallocated
// The value of ptr is undefined
```



?

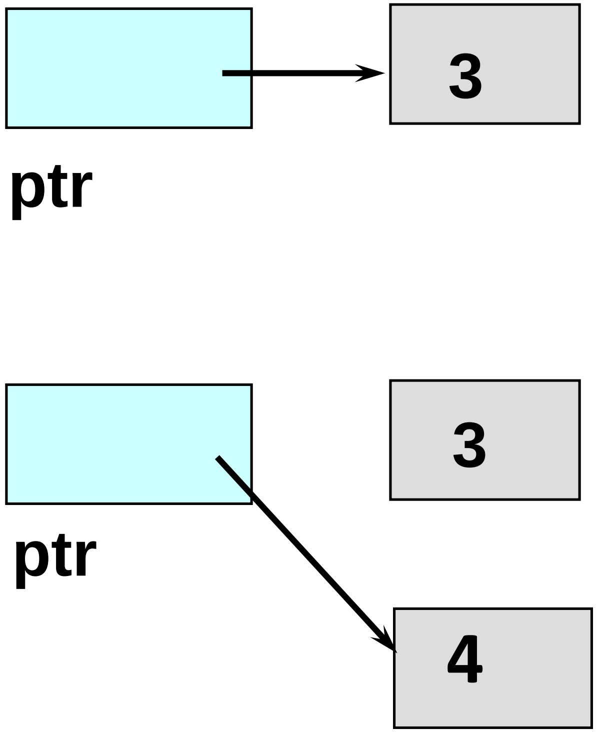
ptr

LECTURE 5

Dangling and Memory Leak

What happens here?

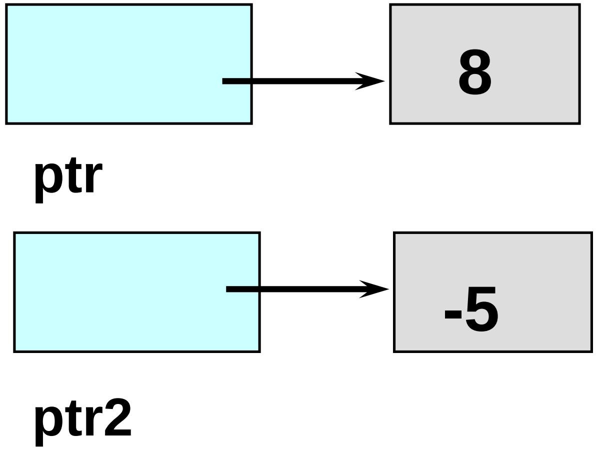
```
int* ptr = new int;
*ptr = 3;

ptr = new int;
// Changes value of ptr
*ptr = 4;
```



Inaccessible Object

An inaccessible object is an unnamed object created by operator `new` that a programmer has left without a pointer to it.

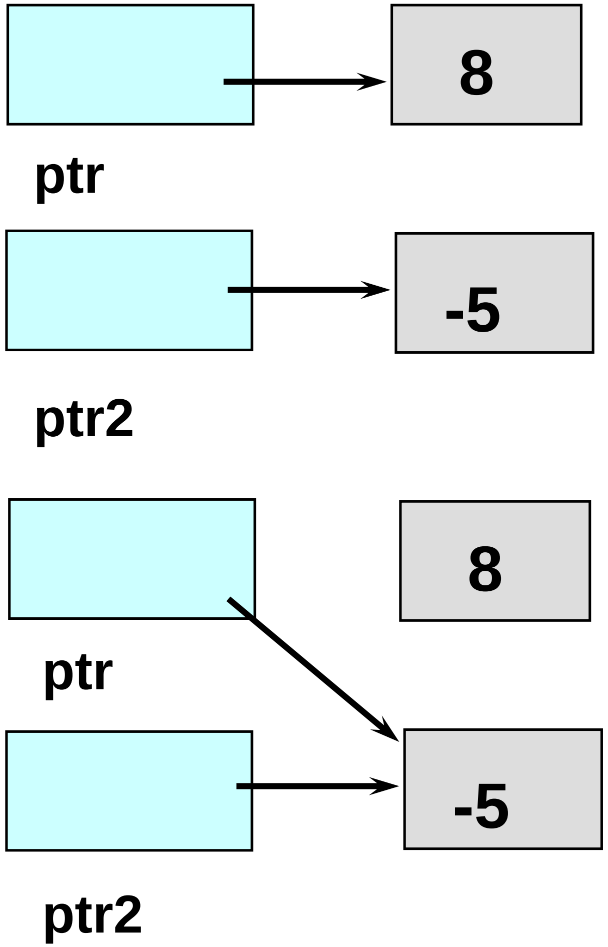
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



How else can an object become inaccessible?

Making an Object Inaccessible

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;

ptr = ptr2;
//Here the 8 becomes
// inaccessible
```



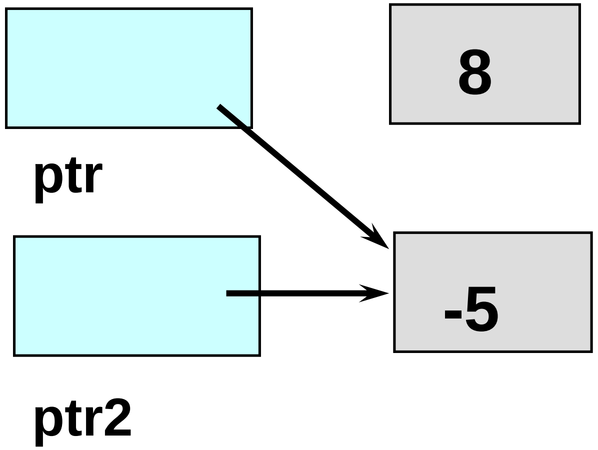
Memory Leak

- A memory leak is the loss of available memory space that occurs when dynamic data is allocated but never deallocated

A Dangling Pointer

- A dangling pointer is a pointer that points to dynamic memory that has been deallocated

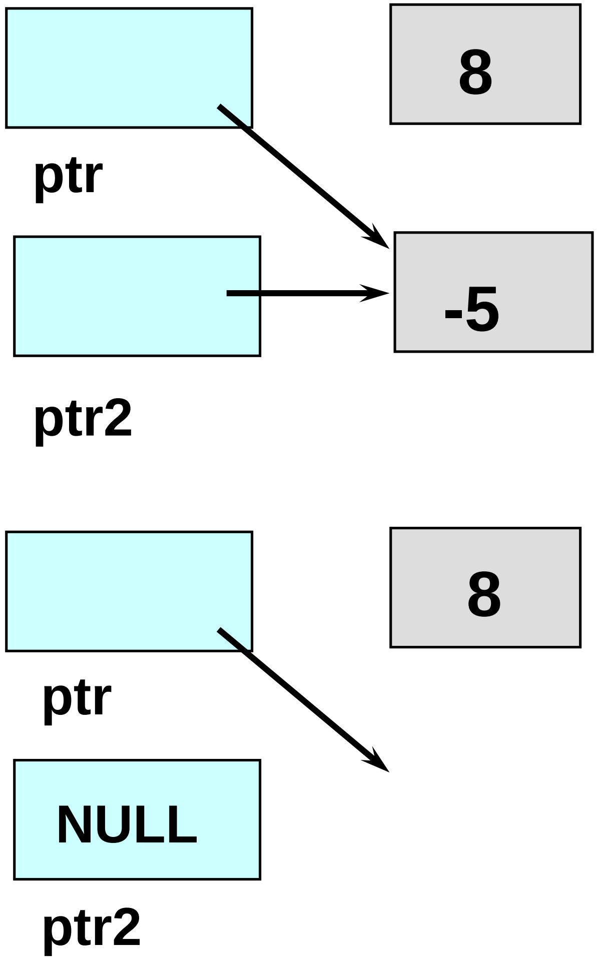
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```



For example,

Leaving a Dangling Pointer

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;

delete ptr2;
// ptr is left dangling
ptr2 = NULL;
```



LECTURE 6

Reference Data Type in C++

Reference Types

int& intRef;

Reference Type Declaration

- DataType& Vaiable;
- DataType &Variable, &Variable, …;

Use reference variable

Use reference variable as ***alias***

```
int someInt;
int& intRef = someInt;
intRef = 1;
void DatePrint(Date& someDate) {
                someDate.Print();
}
```

The difference of using a reference variable and using a pointer variable

Using a Reference Variable

```
int gamma = 26;
int& intRef = gamma;
// Assert: intRef points to gamma
intRef = 35;
// Assert: gamma == 35
intRef = intRef + 3;
// Assert: gamma == 38
```

Using a Pointer Variable

```
int gamma = 26;
int* intPtr = &gamma;
// Assert: intPtr points to gamma
*intPtr = 35;
// Assert: gamma == 35
*intPtr = *intPtr + 3;
// Assert: gamma == 38
```

Reference Variables are *Constant* Pointers

Reference variables cannot be reassigned after initialization

Initialization means:

- Explicit initialization in a declaration
- Implicit initialization by passing an argument to a parameter
- Implicit initialization by returning a function value

The use of reference variables

- The following code fails:

```
void Swap(float x, float y) {
 float temp = x;
 x = y;
 y = temp;
}
Swap(alpha, beta);
```

The use of reference variables

- The following code uses pointer variables:

```
void Swap(float* x, float* y) {
 float temp = *x;
 *x = *y;
 *y = temp;
}
Swap(&alpha, &beta);
```

The use of reference variables

The following code uses reference variables:

```
void Swap(float& x, float& y) {
 float temp = x;
 x = y;
 y = temp;
}
Swap(alpha, beta);
```

The usage of Ampersand (&)

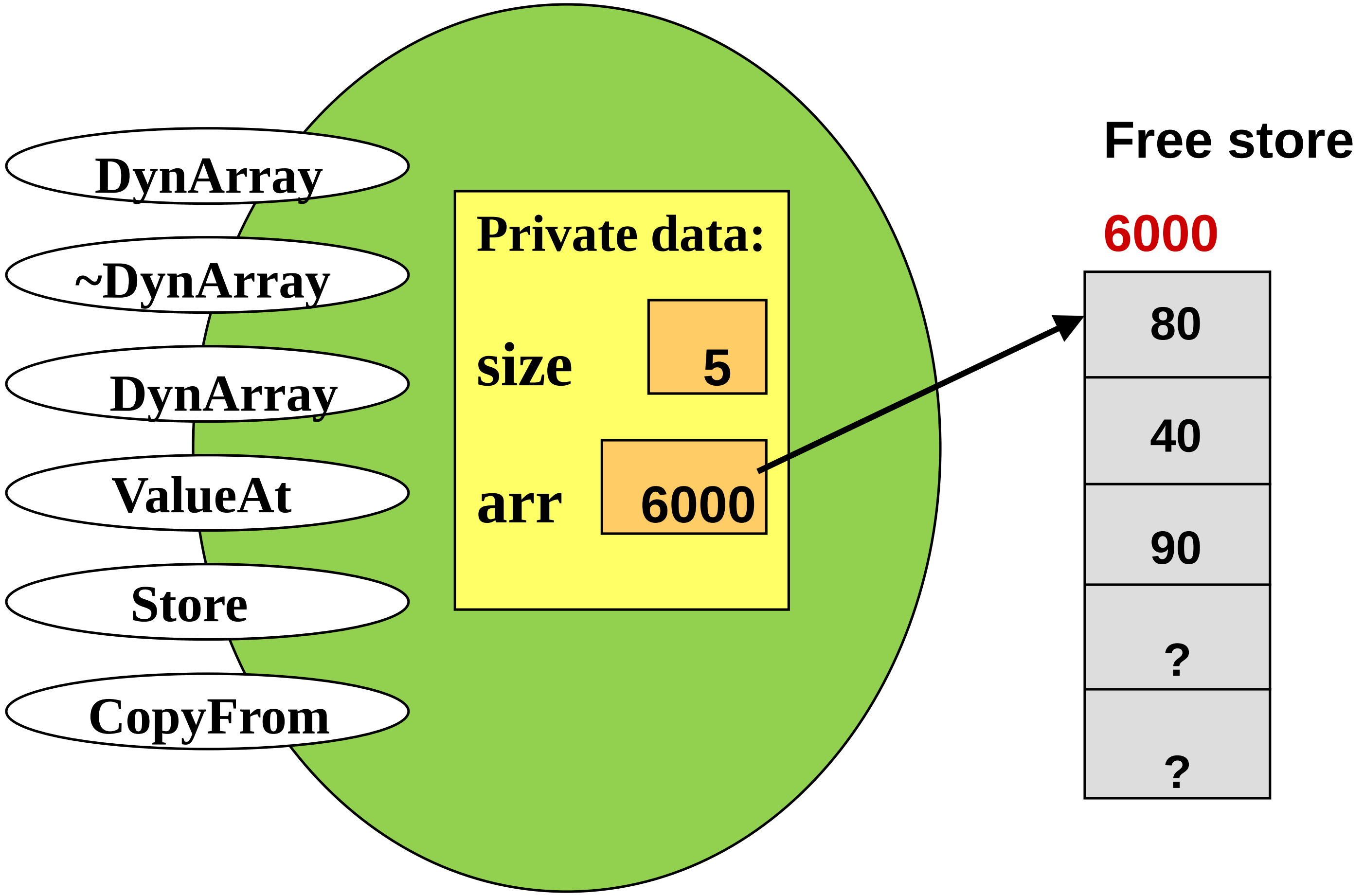
Position	Usage	Meaning
Prefix	&Variable	Address of operation
Infix	Expression & Expression	Bitwise AND operation
Infix	Expression && Expression	Logical AND operation
Postfix	DataType&	DataType (specifically, a reference type) Exception: To declare two variables of reference types, the & must be attached to each variable name: Int &var1, &var2;

LECTURE 7

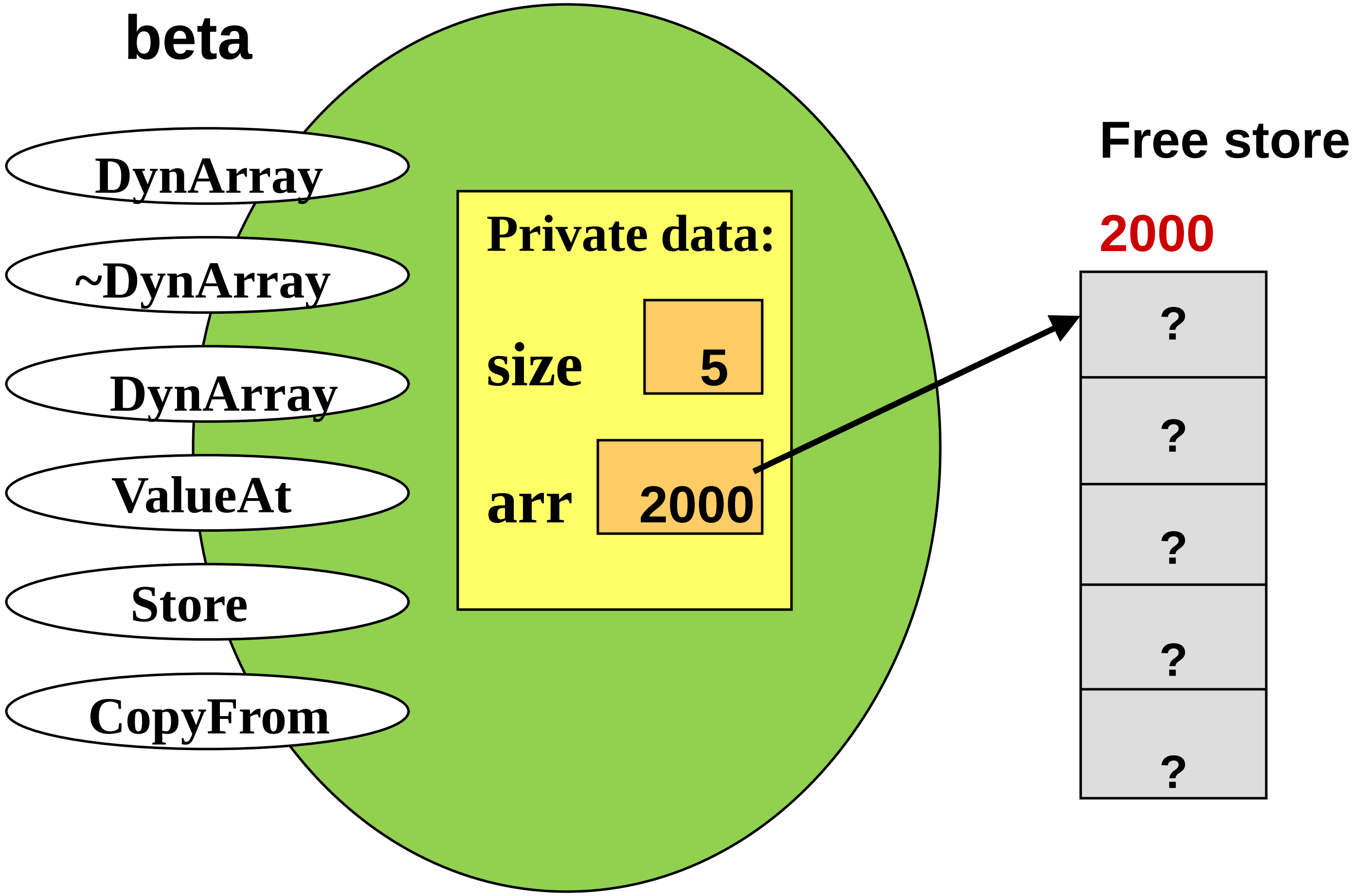
Dynamic Array Class

```
// Specification file (“dynarray.h”)
// Safe integer array class allows run-time specification
// of size, prevents indexes from going out of bounds,
// allows aggregate array copying and initialization

class DynArray{
public:
    DynArray(/* in */  int arrSize);
       // Constructor
        // PRE:  arrSize is assigned
        // POST:  IF arrSize >= 1 && enough memory THEN
        //         Array of size arrSize is created with
           //        all elements == 0  ELSE error message
    DynArray(const DynArray& otherArr);
        // Copy constructor
        // POST: this DynArray is a deep copy of otherArr
        // Is implicitly called for initialization
// Specification file  continued
   ~DynArray();
       // Destructor
       // POST: Memory for dynamic array deallocated

   int  ValueAt (/* in */ int i)  const;
       // PRE:  i is assigned
       // POST: IF 0 <= i < size of this array THEN
       //        FCTVAL == value of array element at index i
       //        ELSE error message

   void  Store (/* in */ int val,  /* in */ int i)
       // PRE:  val and i are assigned
       // POST: IF 0 <= i < size of this array THEN
       //        val is stored in array element i
       //        ELSE error message
// Specification file  continued
    void  CopyFrom (/* in */ DynArray otherArr);
      // POST:  IF enough memory THEN
       //             new array created (as deep copy)
       //             with size and contents
       //             same as otherArr
      //        ELSE error message.
private:
   int*  arr;
   int   size;
};
```

class DynArray

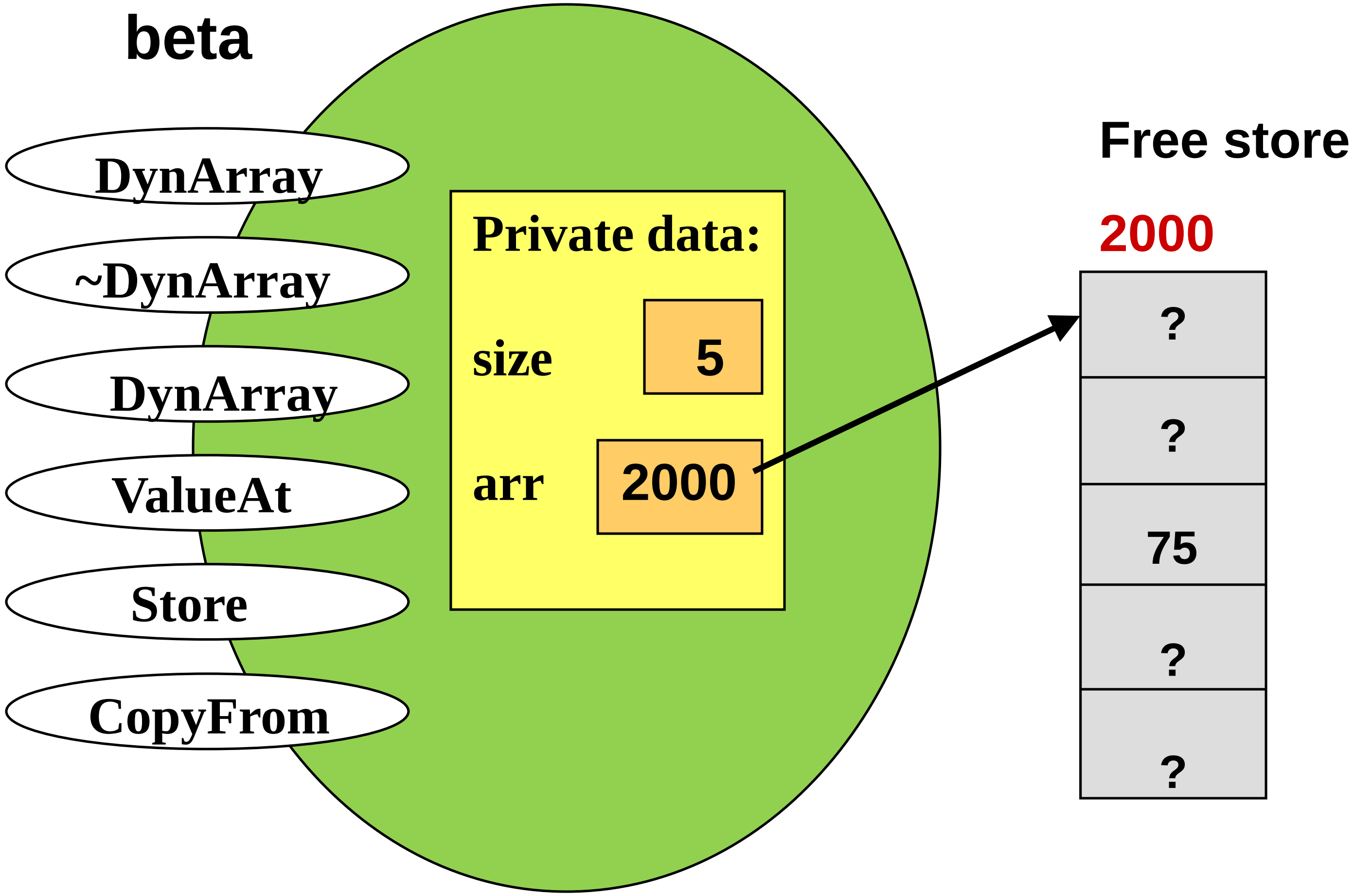


`DynArray beta(5); //constructor`



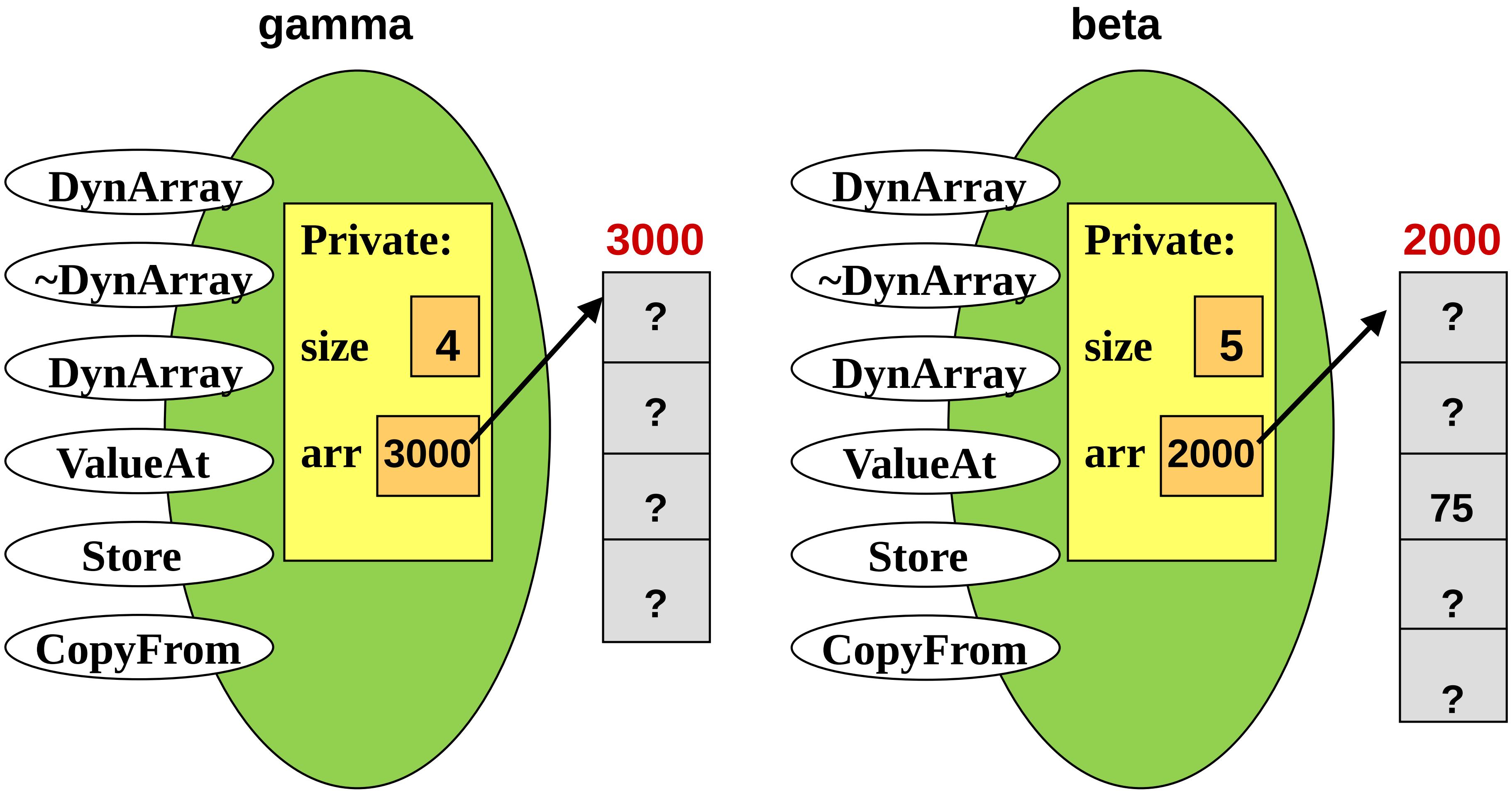
```
DynArray::DynArray(/* in */  int arrSize){
       // Constructor
       // PRE:  arrSize is assigned
       // POST:  IF arrSize >= 1 && enough memory THEN
       //        Array of size arrSize is created with
       //        all elements == 0  ELSE error message
    int i;
    if (arrSize < 1) {
        cerr << “DynArray constructor - invalid size: “
              << arrSize << endl;
        exit(1);
    }

   arr = new  int[arrSize];    // Allocate memory

   size = arrSize;

   for (i = 0; i < size;  i++)arr[i] = 0;
}
```

```
beta.Store(75, 2);
```

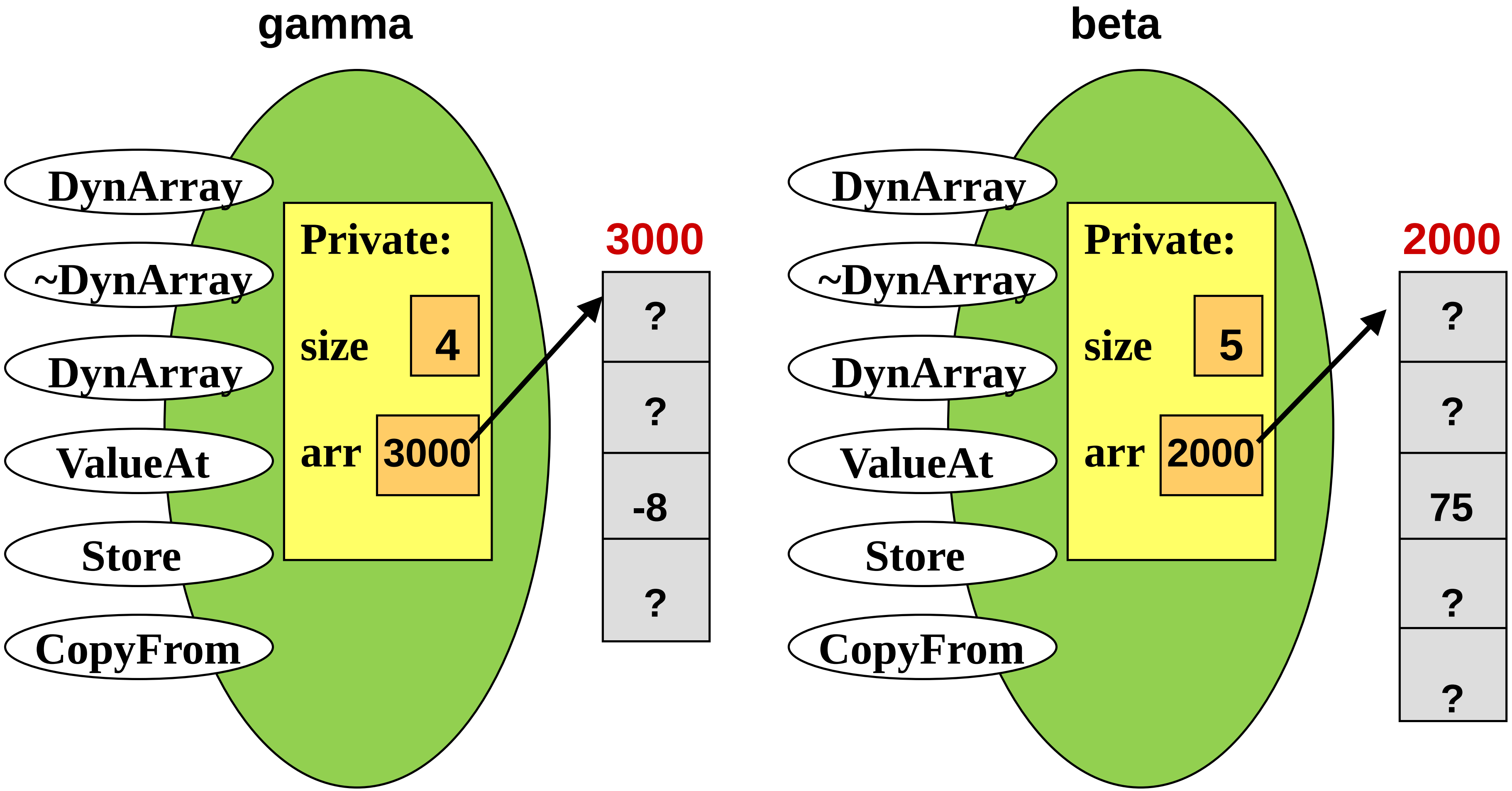


```
void  DynArray::Store (/* in */ int val,  /* in */ int i){

       // PRE:  val and i are assigned
       // POST: IF 0 <= i < size of this array THEN
       //         arr[i] == val
       //         ELSE error message
   if (i < 0 || i >= size) {
         cerr << “Store - invalid index : ” << i << endl;
         exit(1);
    }
 arr[i] = val;
}
```

`DynArray gamma(4);//Constructor`



`gamma.Store(-8,2);`



```
int  DynArray::ValueAt (/* in */ int i)  const {
      // PRE:  i is assigned
      // POST: IF 0 <= i < size THEN
      //       Return value == arr[i]
      //       ELSE halt with error message
    if (i < 0 || i >= size) {
        cerr << “ValueAt - invalid index : “ << i
           << endl;
        exit(1);
    }
    return arr[i];
}
```

Why is a destructor needed?

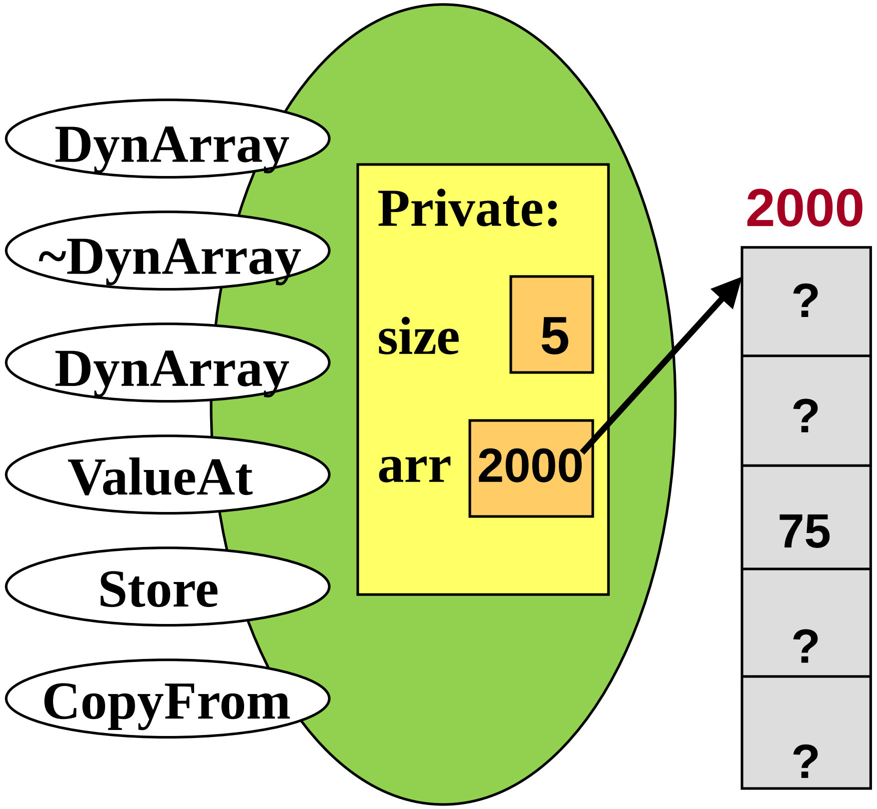
- When a DynArray class variable goes out of scope, the memory space for data members size and pointer `arr` is deallocated
- But the dynamic array that `arr` points to is not automatically deallocated
- A class destructor is used to deallocate the dynamic memory pointed to by the data member

class DynArray Destructor

```
DynArray::~DynArray()
 // Destructor
 // POST: Memory for dynamic array deallocated
{
     delete [ ] arr;
}
```

What happens . . .

- When a function is called that passes a DynArray object by value, what happens?

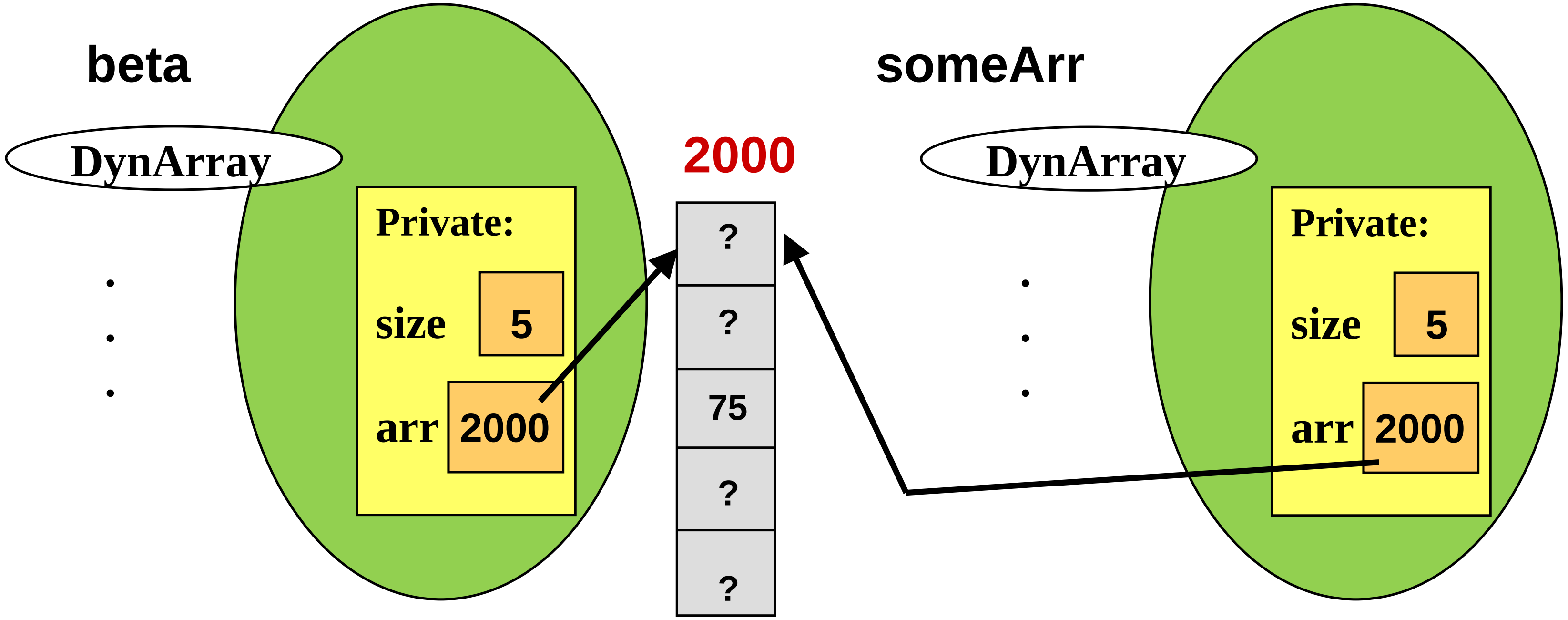


Passing a Class Object by Value

```
// Function code

void  SomeFunc(DynArray  someArr)
 // Uses pass by value
{
       .
             .
       .
       .
}
```

By default,
Pass-by-value makes a shallow copy

```
    DynArray  beta(5);          // Client code
          .
          .
          .
    SomeFunc(beta);           //  Function call
```



shallow copy

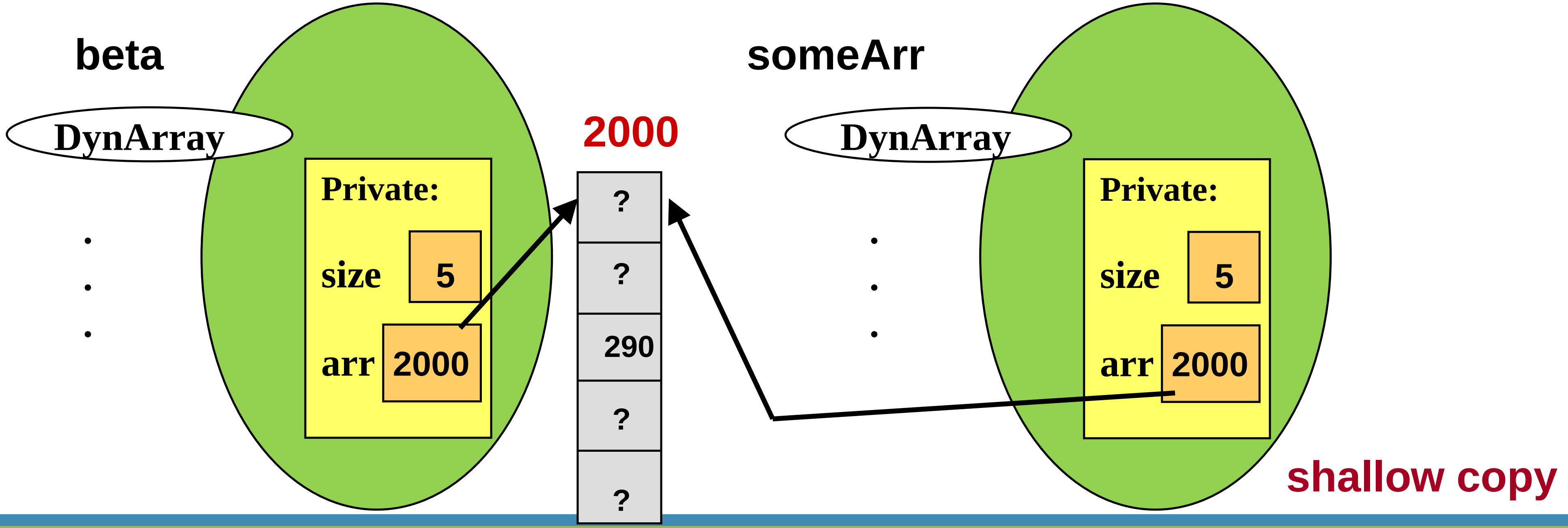
Suppose SomeFunc calls Store

```
void  SomeFunc(DynArray  someArr)
 // Uses pass by value
{
       someArr.Store(290, 2);
              .
       .
       .

}
```

What happens in the shallow copy scenario?

`beta.arr[2]` has changed

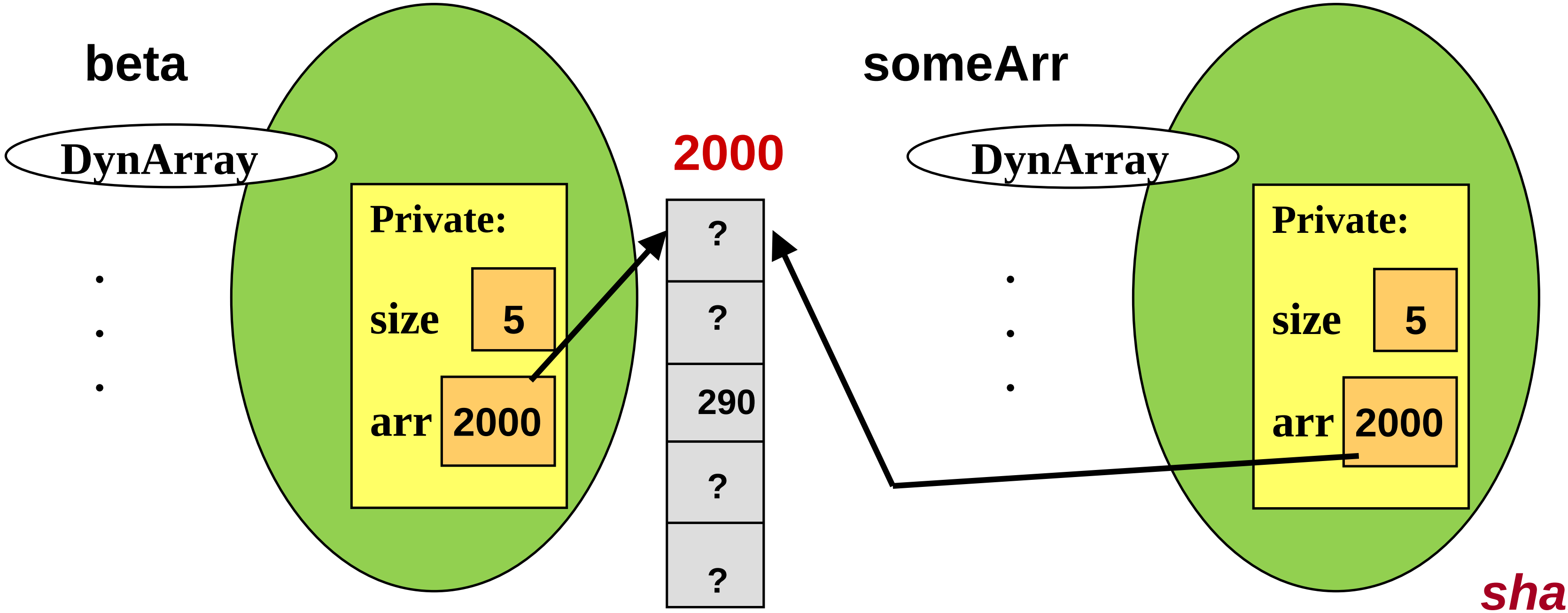
```
DynArray  beta(5);        // Client code
       .
       .
       .
SomeFunc(beta);
```



shallow copy

`beta.arr[2]` has changed

Although beta is passed by value, its dynamic data has changed!



shallow copy

LECTURE 8

Deep Copy and Shallow Copy

Shallow Copy vs. Deep Copy

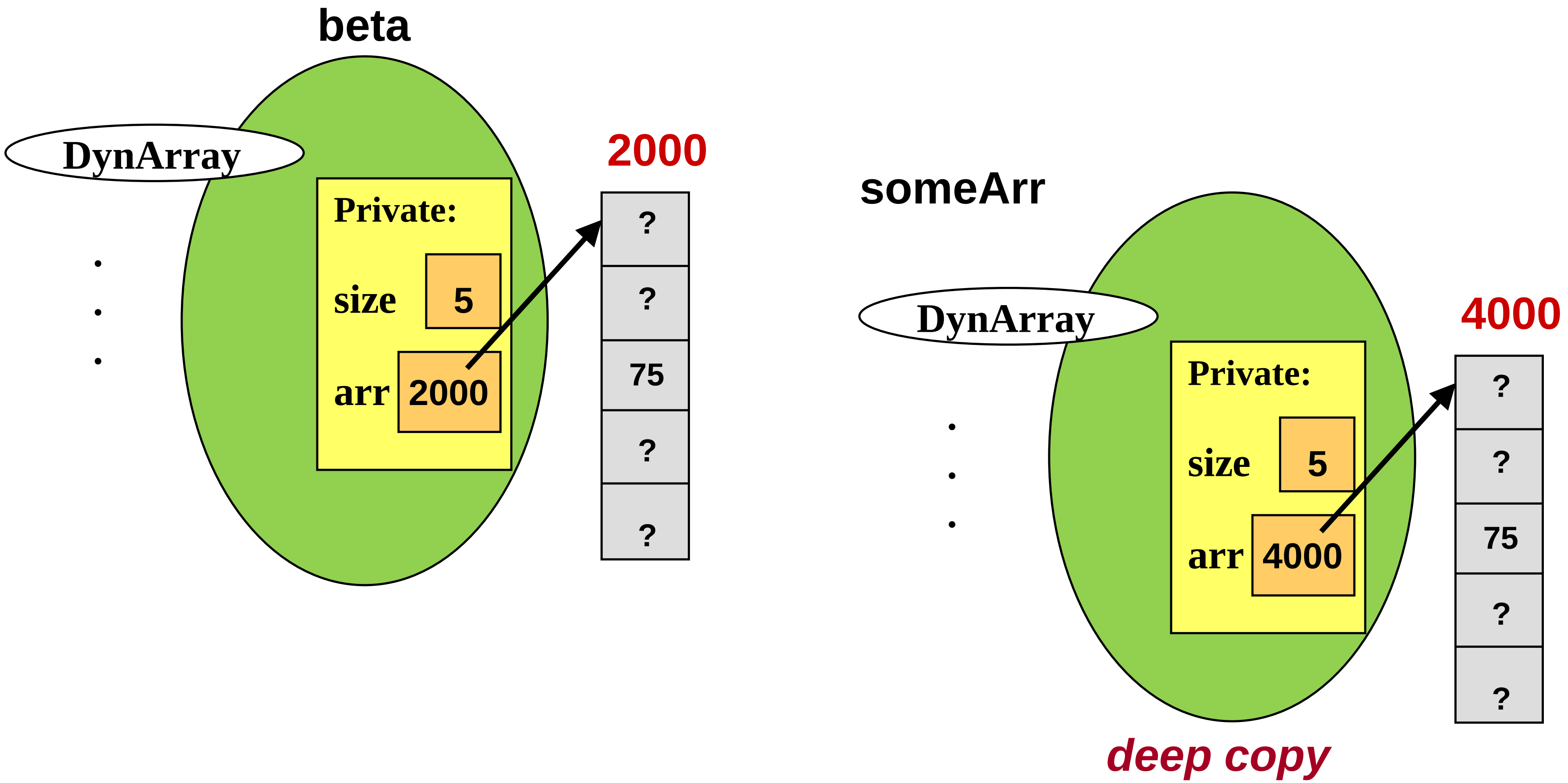
- *A shallow copy* copies only the class data members, and does not make a copy of any pointed-to data

- *A deep copy* copies not only the class data members, but also makes a separate stored copy of any pointed-to data

What’s the difference?

- *A shallow copy* shares the pointed to dynamic data with the original class object
- *A deep copy* makes its own copy of the pointed to dynamic data at different locations than the original class object

Making a (Separate) Deep Copy



LECTURE 9

Copy Constructor

Initialization of Class Objects

- C++ defines initialization to mean
  - initialization in a variable declaration
  - passing an object argument by value
  - returning an object as the return value of a function
- By default, C++ uses shallow copies for these initializations

As a result . . .

- When a class has a data member that points to dynamically allocated data, you must write what is called a copy constructor
- The copy constructor is implicitly called in initialization situations and makes a deep copy of the dynamic data in a different memory location

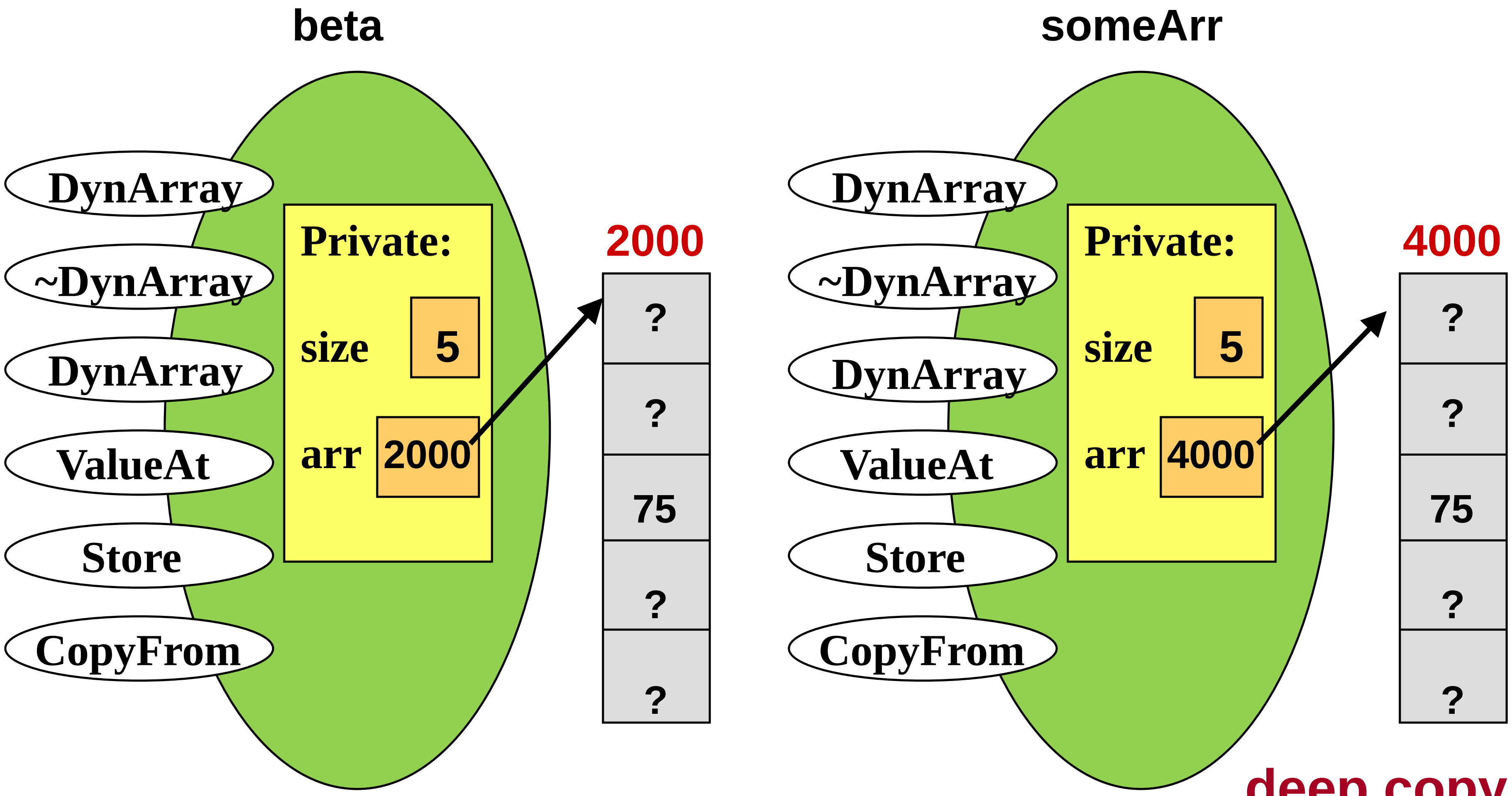
More about Copy Constructors

- When you provide (write) a copy constructor for a class, the copy constructor is used to make copies for pass by value
- You do not explicitly call the copy constructor
- Like other constructors, it has no return type
- Because the copy constructor properly defines pass by value for your class, it must use pass by reference in its definition

Copy Constructor

- Copy constructor is a special member function of a class that is implicitly called in these 3 situations:
  - Passing object parameters by value
  - Initializing an object variable in its declaration
  - Returning an object as the return value of a function

```
SomeFunc(beta);  // copy-constructor
                 // beta passed by value
```



deep copy

Classes with Data Member Pointers Need

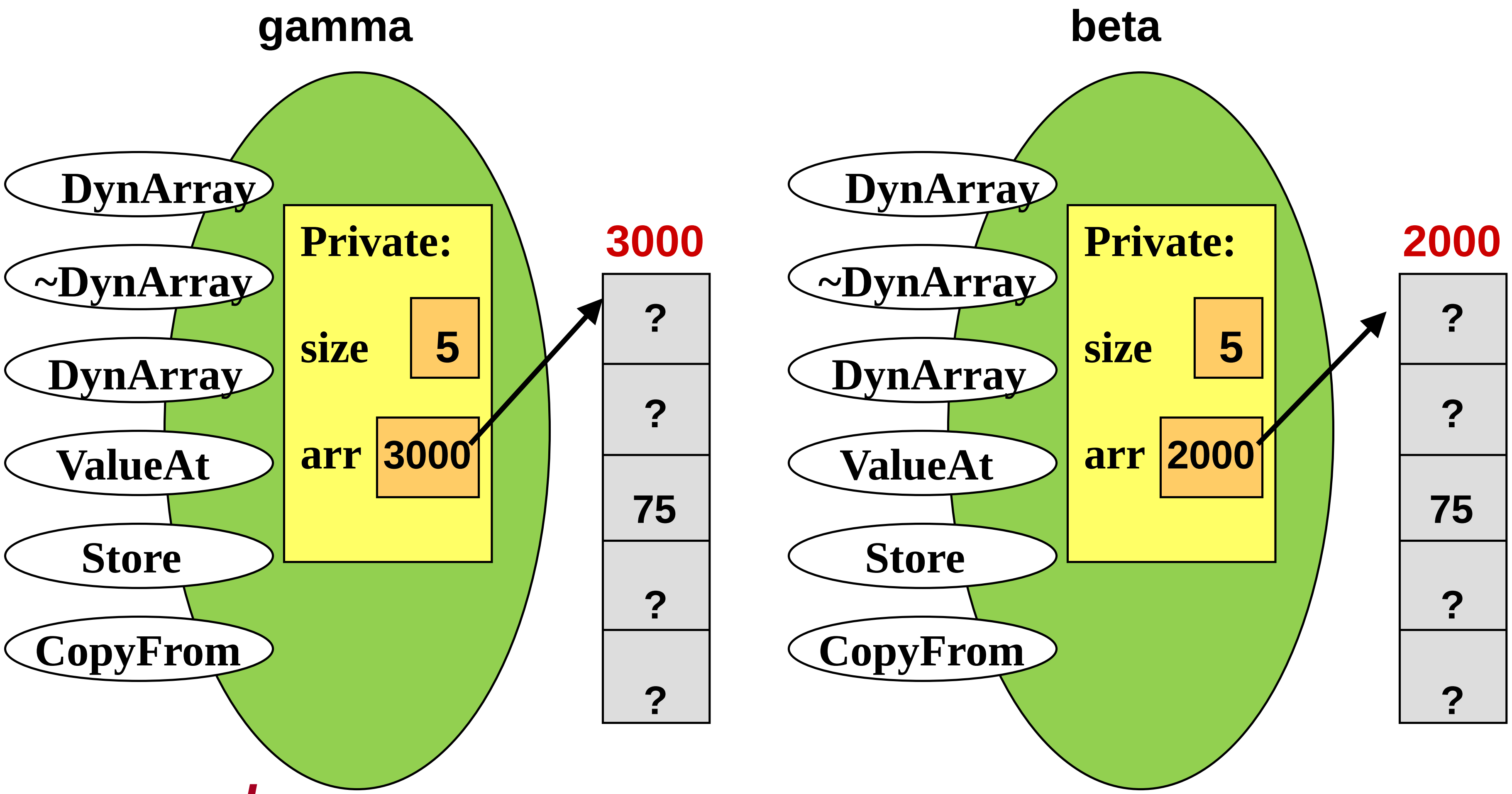
1. Constructor
2. Copy Constructor
3. Destructor

```
DynArray::DynArray(const DynArray& otherArr){
      // Copy constructor
      // Implicitly called for deep copy in initializations
      // POST:  If room on free store THEN
      //     new array of size otherArr.size is created
      //     on free store && arr == its base address
      //     && size == otherArr.size
      //     && arr[0..size-1] == otherArr.arr[0..size-1]
      //     ELSE error occurs
    int i;
    size = otherArr.size;
    arr = new  int[size];    // Allocate memory for copy
    for (i = 0; i< size; i++)
        arr[i] = otherArr.arr[i];     // Copies array
}
```

What about the assignment operator?

- The default method used for assignment of class objects makes a shallow copy
- If your class has a data member that points to dynamic data, you should write a member function to create a deep copy of the dynamic data

gamma.CopyFrom(beta);

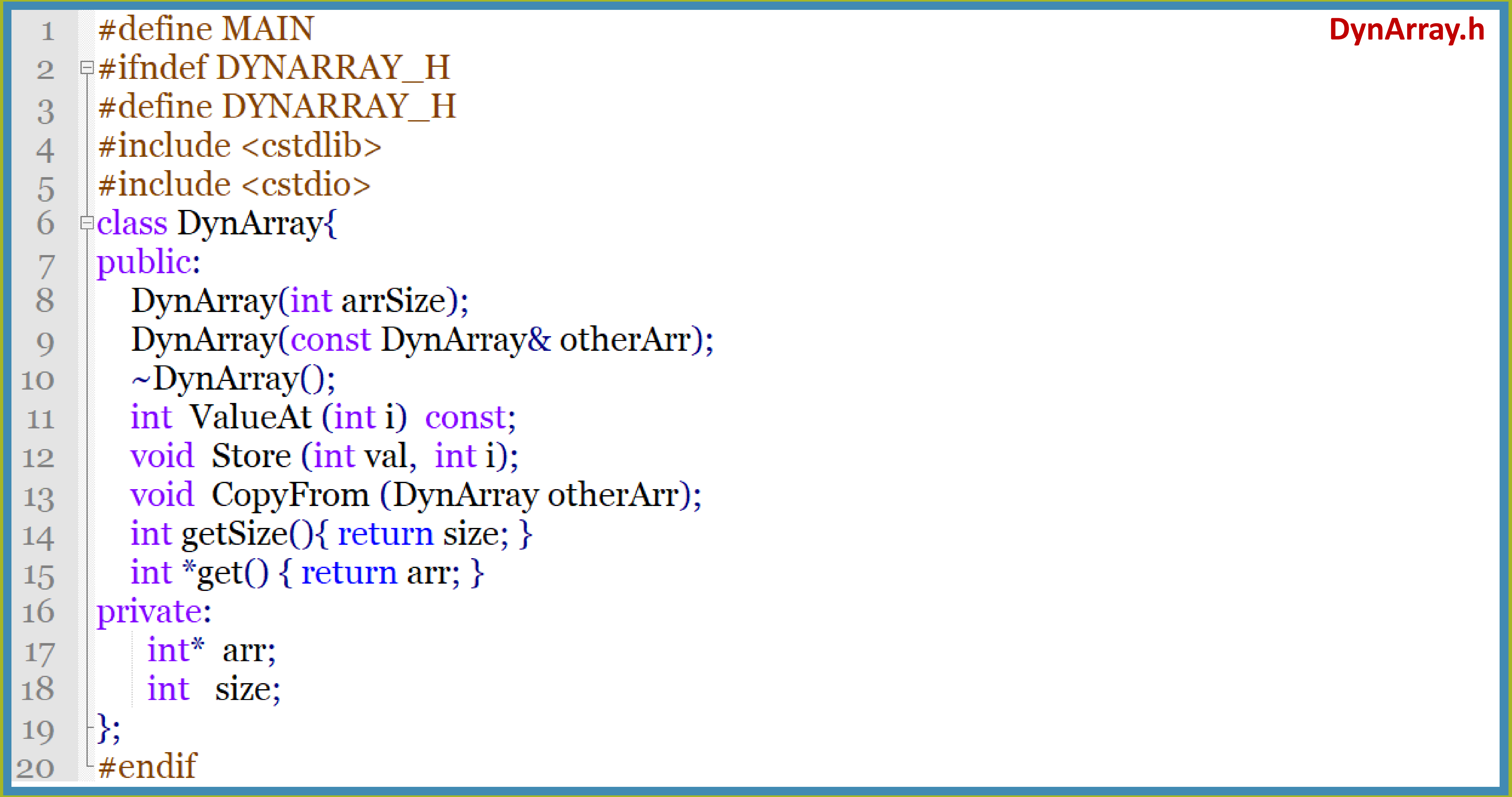


deep copy

```
void  DynArray::CopyFrom (/* in */ DynArray  otherArr)
        // Creates a deep copy of otherArr
        // POST:  Array pointed to by arr@entry deallocated
        //   &&  IF room on free store
        //        THEN new array is created on free store
        //           && arr == its base address
        //           && size == otherArr.size
        //           && arr[0..size-1] == otherArr[0..size-]
        //       ELSE halts with error message
{
    int i;

    delete[ ]  arr;             // Delete current array
    size = otherArr.size;
    arr = new int [size];               // Allocate new array

    for (i = 0; i< size; i++)  // Deep copy array
        arr[i] = otherArr.arr[i];
}
```

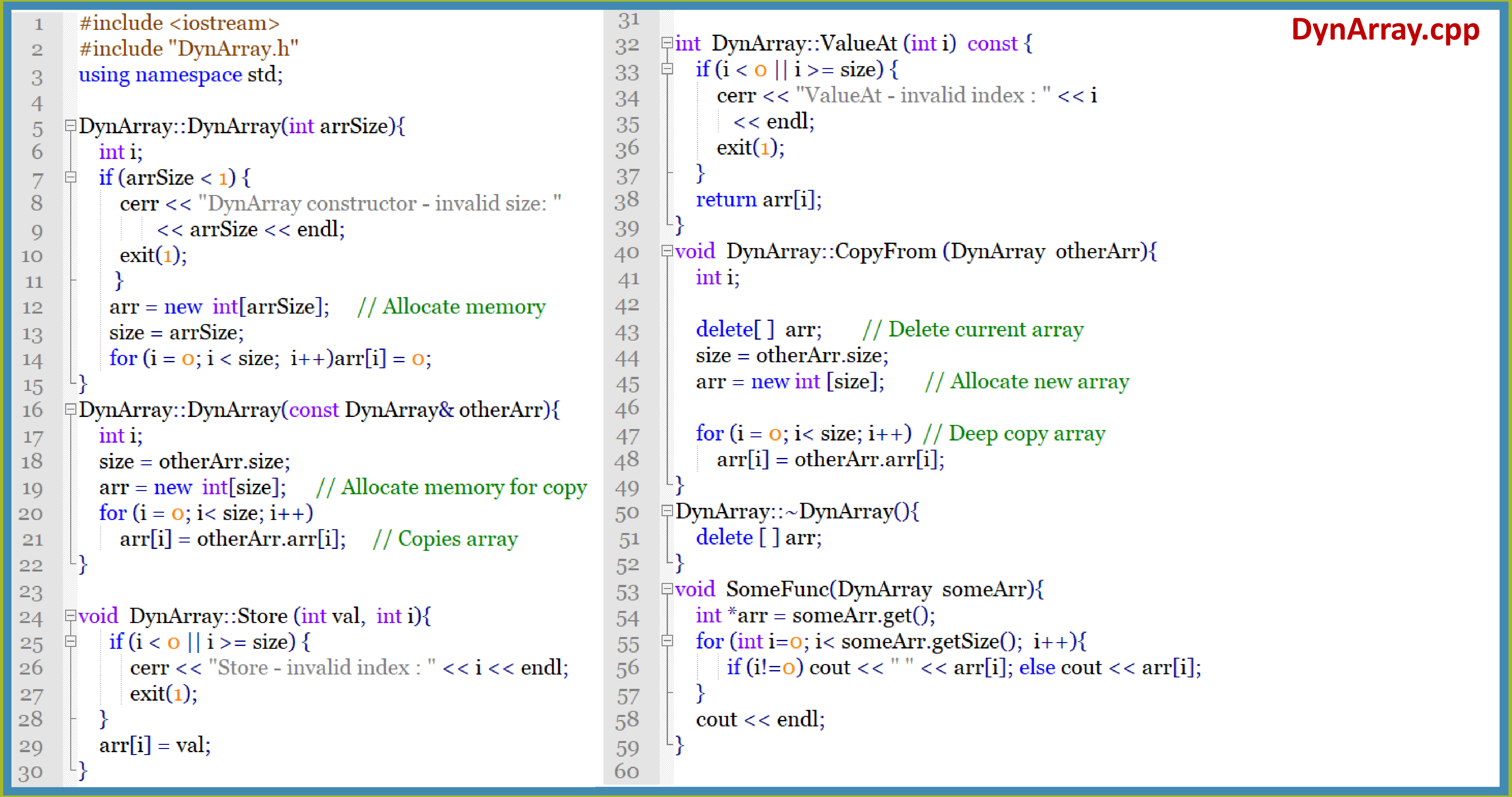
Demo Program: DynArray.cpp

Go Notepad++!!!



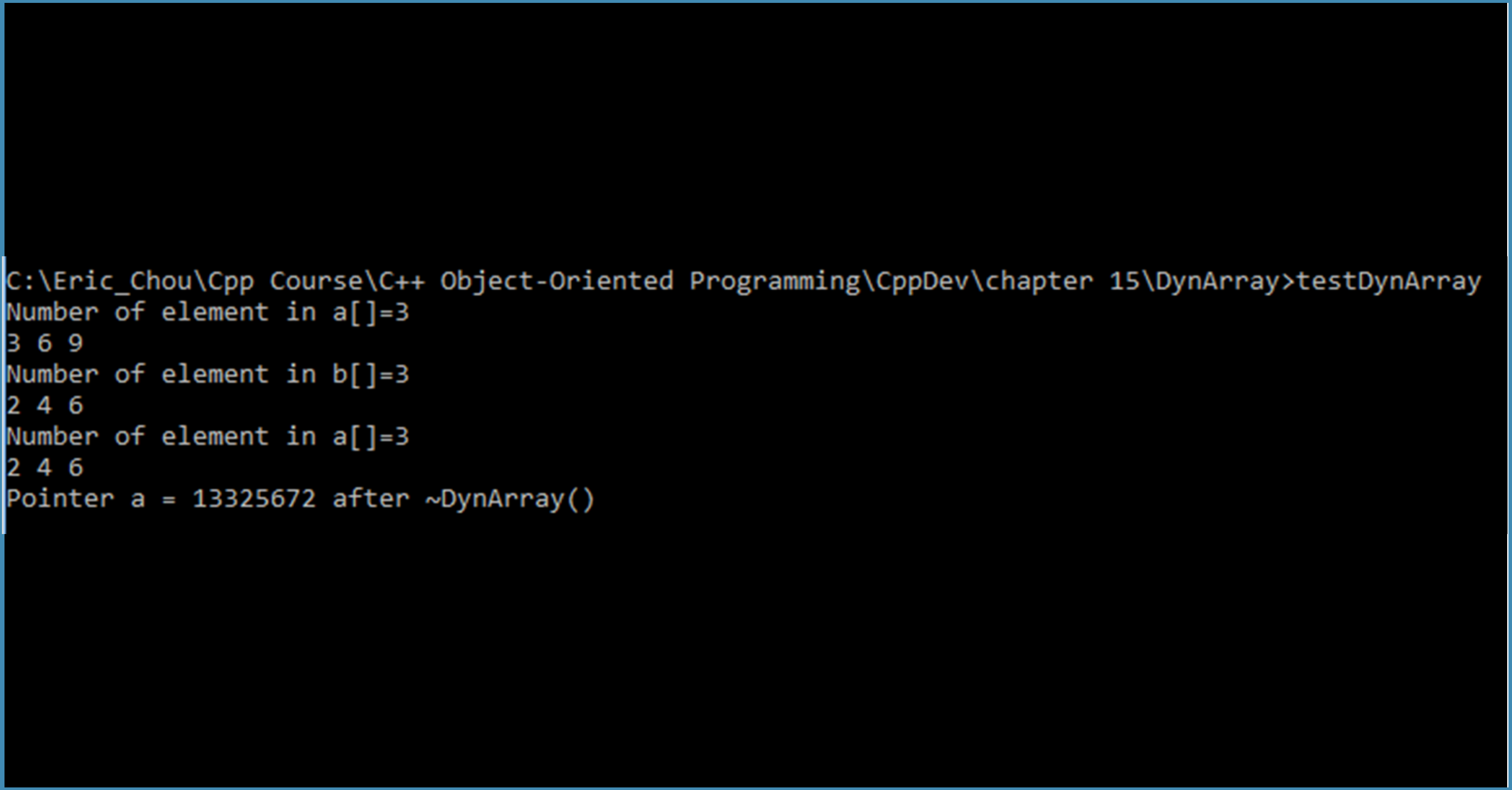
DynArray.h

```
#define MAIN
#ifndef DYNARRAY_H
#define DYNARRAY_H
#include <cstdlib>
#include <cstdio>
class DynArray{
public:
    DynArray(int arrSize);
    DynArray(const DynArray& otherArr);
    ~DynArray();
    int  ValueAt (int i)  const;
    void  Store (int val,  int i);
    void  CopyFrom (DynArray otherArr);
    int getSize(){ return size; }
    int *get() { return arr; }
private:
      int*  arr;
      int   size;
};
#endif
```



DynArray.cpp

```cpp
#include <iostream>
#include "DynArray.h"
using namespace std;

DynArray::DynArray(int arrSize){
    int i;
    if (arrSize < 1) {
        cerr << "DynArray constructor - invalid size: "
             << arrSize << endl;
        exit(1);
    }
    arr = new  int[arrSize];     // Allocate memory
    size = arrSize;
    for (i = 0; i < size;  i++)arr[i] = 0;
}
DynArray::DynArray(const DynArray& otherArr){
    int i;
    size = otherArr.size;
    arr = new  int[size];     // Allocate memory for copy
    for (i = 0; i< size; i++)
        arr[i] = otherArr.arr[i];     // Copies array
}

void  DynArray::Store (int val,  int i){
    if (i < 0 || i >= size) {
        cerr << "Store - invalid index : " << i << endl;
        exit(1);
    }
    arr[i] = val;
}

int  DynArray::ValueAt (int i)  const {
    if (i < 0 || i >= size) {
        cerr << "ValueAt - invalid index : " << i
            << endl;
        exit(1);
    }
    return arr[i];
}
void  DynArray::CopyFrom (DynArray  otherArr){
    int i;

    delete[ ]  arr;       // Delete current array
    size = otherArr.size;
    arr = new int [size];       // Allocate new array

    for (i = 0; i< size; i++)  // Deep copy array
        arr[i] = otherArr.arr[i];
}
DynArray::~DynArray(){
    delete [ ] arr;
}
void  SomeFunc(DynArray  someArr){
    int *arr = someArr.get();
    for (int i=0; i< someArr.getSize();  i++){
        if (i!=0) cout << " " << arr[i]; else cout << arr[i];
    }
    cout << endl;
}
```



```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 15\DynArray>testDynArray
Number of element in a[]=3
3 6 9
Number of element in b[]=3
2 4 6
Number of element in a[]=3
2 4 6
Pointer a = 13325672 after ~DynArray()
```