CPE 212 - Fundamentals of Software Engineering

Pointers

Reminder:

Project 02 due Friday January 31 by 11:59pm

Objective:
Introduction into the wonderful world of pointers.

Outline

- Direct vs Indirect Addressing
- Examples
- Pointers, Structs and Arrays
- Static vs Dynamic Allocation
- Deallocation

Direct vs. Indirect Addressing

- Direct Addressing
  - Accessing a variable in one-step by using the variable name
- Indirect Addressing
  - Accessing a variable in two-steps by first using a pointer that gives the location of the variable

Pointers

- Pointer
  - A variable which contains the address or location of other variables
- Pointer declaration syntax:

```
DataType* VariableName;
        or
DataType *Variable1, *Variable2...;
```

Warning

```
int * p, q;     // p is a pointer, q is not
```

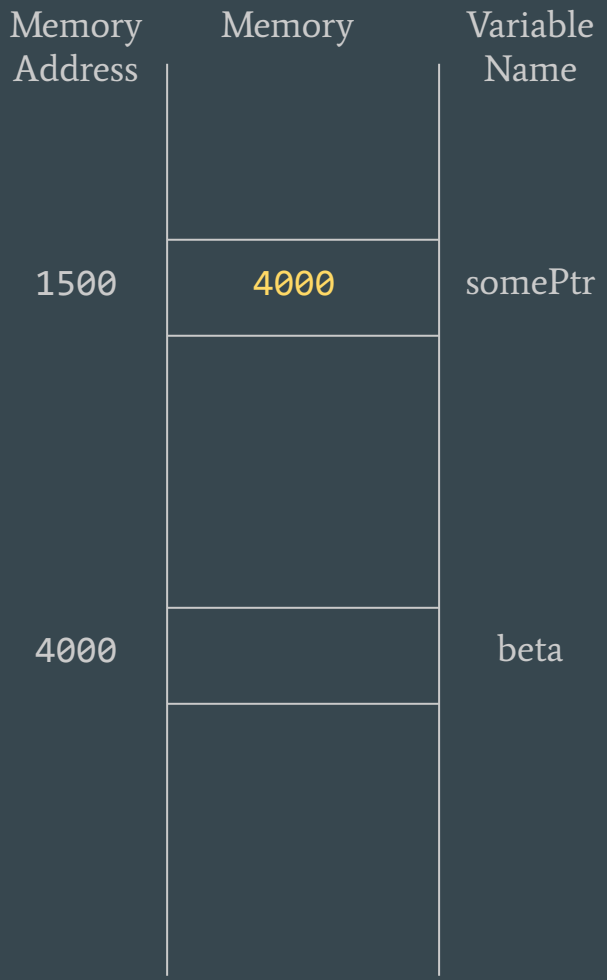
Pointers in Memory

```
int   beta; // Normal int variable
int*  somePtr;  // Pointer declaration
```

NOTE: Memory Address are typically written as hex values. Decimals are used for simplicity

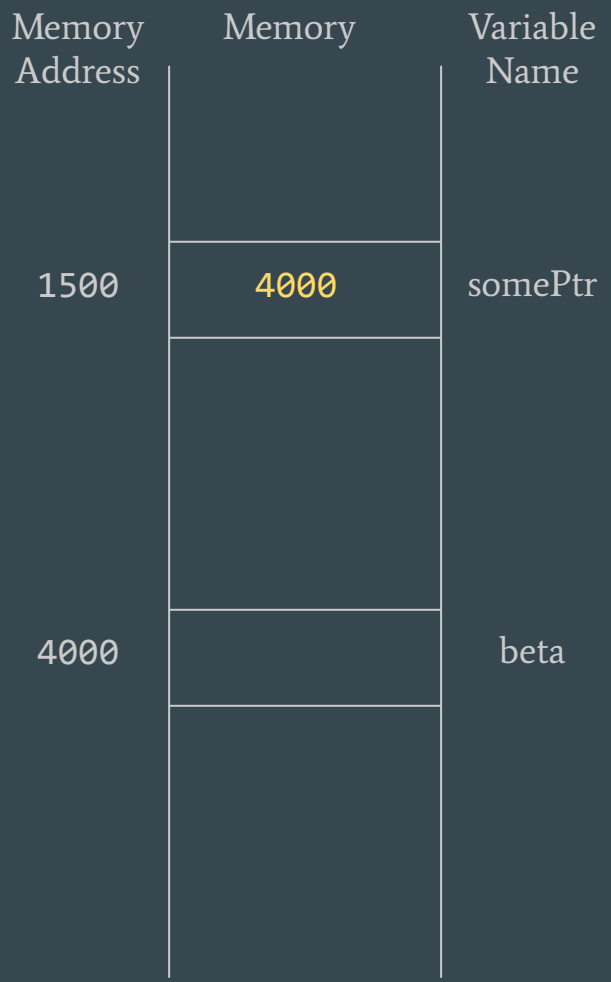
Pointers in Memory

```
int   beta;     // Normal int variable
int*  somePtr;  // Pointer declaration

somePtr = &beta;    // Address-of operator
```



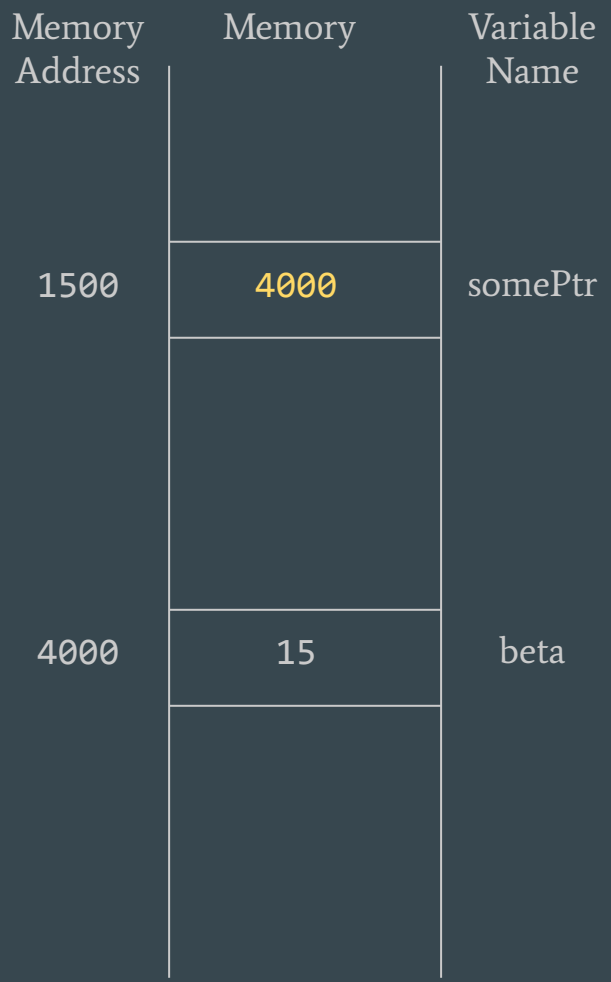
Pointers in Memory

```
int   beta;     // Normal int variable
int*  somePtr;  // Pointer declaration

somePtr = &beta;    // Address-of operator

// int* somePtr = &beta; gives same result
```



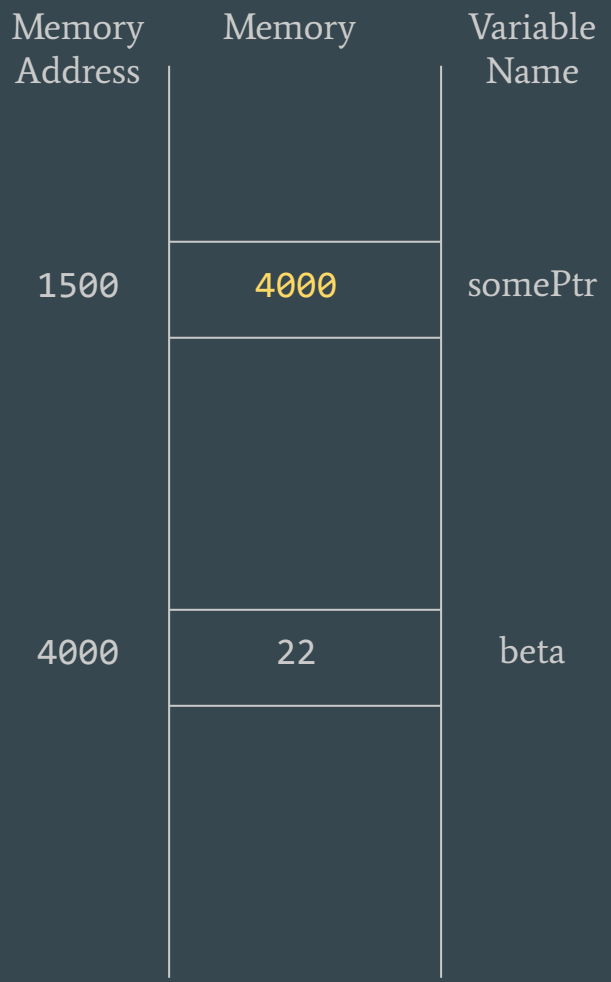
Pointers in Memory

```
int   beta;     // Normal int variable
int*  somePtr;  // Pointer declaration

somePtr = &beta;    // Address-of operator

// int* somePtr = &beta; gives same result

beta = 15;  // Direct addressing
```



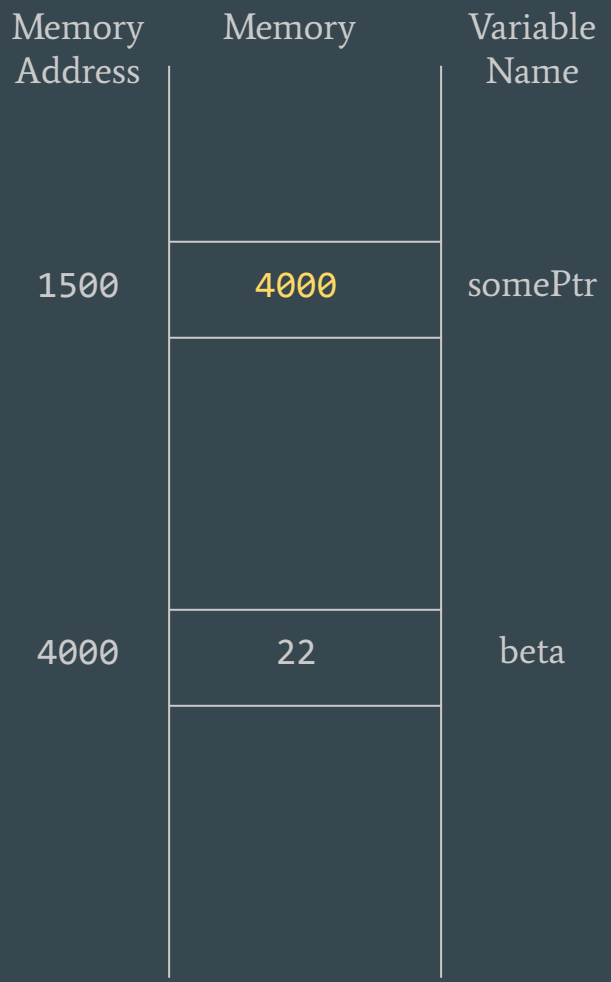
Pointers in Memory

```
int   beta;     // Normal int variable
int*  somePtr;  // Pointer declaration

somePtr = &beta;    // Address-of operator

// int* somePtr = &beta; gives same result

beta = 15;  // Direct addressing

*somePtr = 22;  // Indirect addressing using
                // dereference operator
```



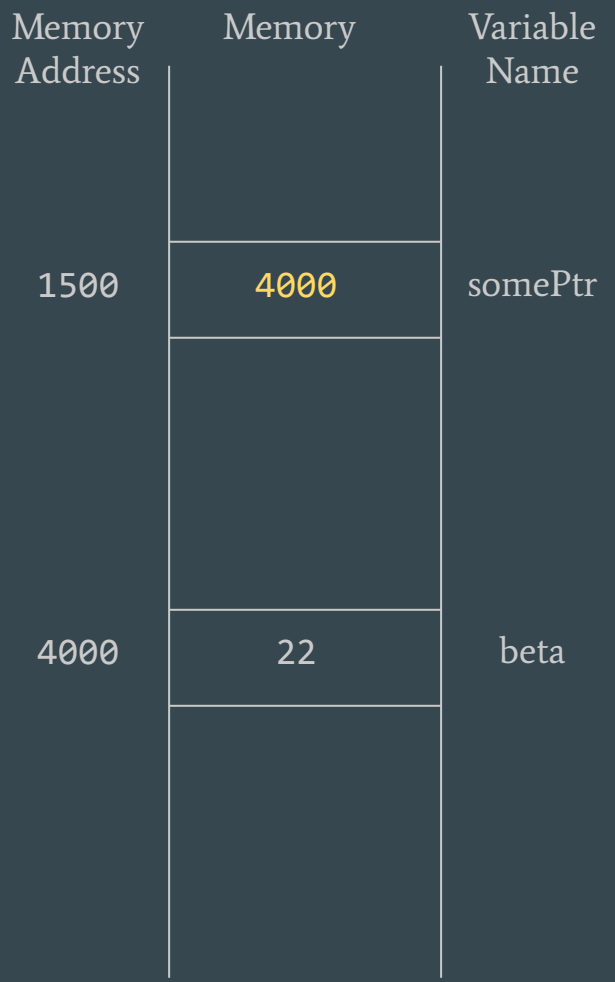
Pointers in Memory

```
int   beta;     // Normal int variable
int*  somePtr;  // Pointer declaration

somePtr = &beta;    // Address-of operator

// int* somePtr = &beta; gives same result

beta = 15;  // Direct addressing

*somePtr = 22;  // Indirect addressing using
                // dereference operator
// Pointer Questions...
cout << somePtr << endl;  // Prints what??
```



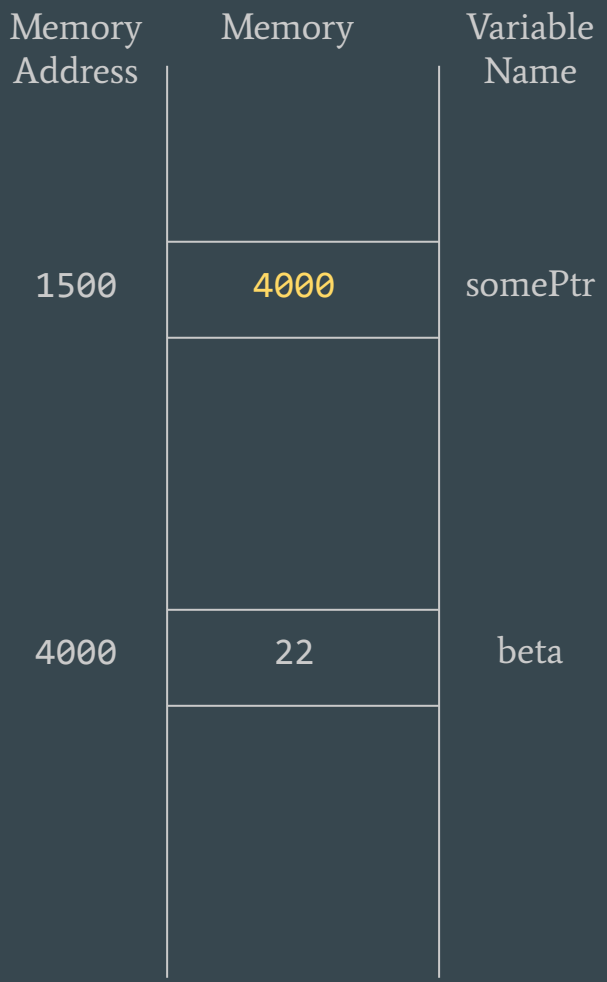
Pointers in Memory

```
int   beta;     // Normal int variable
int*  somePtr;  // Pointer declaration

somePtr = &beta;    // Address-of operator

// int* somePtr = &beta; gives same result

beta = 15;  // Direct addressing

*somePtr = 22;  // Indirect addressing using
                // dereference operator
// Pointer Questions...
cout << *somePtr << endl;  // Prints what??
```



Pointers in Memory

```
int   beta;     // Normal int variable
int*  somePtr;  // Pointer declaration

somePtr = &beta;    // Address-of operator &

// int* somePtr = &beta; gives same result

beta = 15;      // Direct addressing

*somePtr = 22;  // Indirect addressing
          // using dereference op *

cout << somePtr << endl;  // Prints 4000
cout << *somePtr << endl; // Prints 22
cout << &somePtr << endl; // Prints 1500

// More Pointer Questions...
(*somePtr)++;   // Increments the variable by one
cout << somePtr << endl;  // Prints 4000
cout << *somePtr << endl; // Prints 23
cout << &somePtr << endl; // Prints 1500

somePtr++;      // Be very careful here!!!
// The compiler knows that somePtr points to an integer value
// So, the compiler generates the machine language code that
// implements the following calculation
// somePtr = somePtr + sizeof(int);
cout << somePtr << endl;  // Prints 4004
```



Pointers, Structs, and Arrays

```
struct StudentRec
{
   string  firstName;
   string  lastName;
   float     gpa;
};

StudentRec  someStudent;
StudentRec* studentRecPtr;
studentRecPtr = &someStudent;

(*studentRecPtr).firstName = “Homer”;
(*studentRecPtr).lastName = “Simpson”;
// Note: Parentheses required above due to operator precedence rules

studentRecPtr->gpa = 0.4;   // Combines pointer dereference and member selection operator
// Equivalent to (*studentRecPtr).gpa = 0.4;

StudentRec* myStudentPtrs[10];  // Array of ten StudentRec pointers
// Note: none of these pointers have been initialized above

myStudentPtrs[1] = &someStudent;
myStudentPtrs[1]->lastName = “Simpson”;
myStudentPtrs[1]->firstName = “Bart”;
myStudentPtrs[1]->gpa = 2.1;
```

Pointers, Structs, and Arrays

```
int   myArray[10]; // Normal int array variable
int*  somePtr;    // Pointer declaration

somePtr = &myArray[0];  // Sets somePtr equal to base address of myArray

somePtr = myArray;  // Has same effect. Why?
```

Pointers, Structs, and Arrays

```
int   myArray[10];    // Normal int array variable
int*  somePtr;    // Pointer declaration

somePtr = &myArray[0];  // Sets somePtr equal to base address of myArray

somePtr = myArray;      // Has same effect. Why?


void SomeFunction(int someArray[])
{
   // Useful code here...
   someArray[0] = 4;
}

// Can rewrite the above function definition as follows
void SomeFunction(int* someArray)
{
   // Useful code here...
   someArray[0] = 4;
}
```

Static vs Dynamic Allocation

- Static Allocation
  - Performed at compile time

- Dynamic Allocation
  - Allocates memory at run time

Question?

Where does memory come from?

Dynamic Allocation Operator *new*

```
int*  someIntPtr;       // Reserves memory for pointer variable

someIntPtr = new int;   // new Allocates memory to hold an int and
                        // stores the address into someIntPtr

// new creates an unnamed, uninitialized variable of the designated type
// and returns a pointer to this variable
//
// What happens if computer has run out of available memory?
//   An exception will occur or new returns NULL pointer depending upon
compiler

char* baseAddress;

baseAddress = new char[20]; // Creates 20 element char array and places its
                            // base address into baseAddress
```

Dynamic Allocation

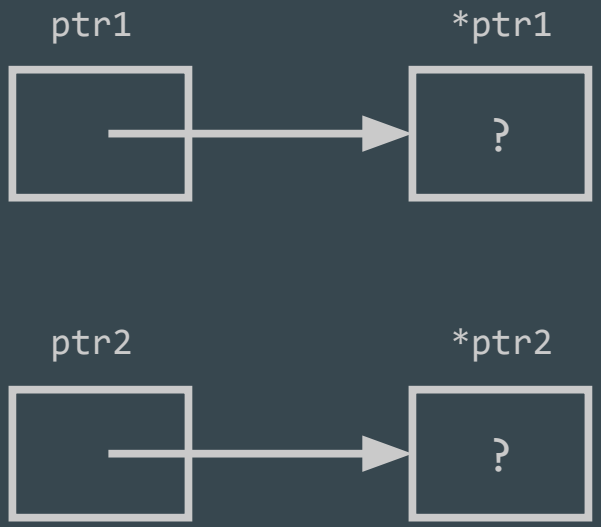
- Memory Leak
  - Loss of available memory space when that memory is dynamically allocated but not deallocated
- Garbage
  - Memory locations that can no longer be accessed
- Inaccessible Object
  - A dynamic variable on the free store without any pointer pointing to it
- Dangling Pointer
  - A pointer that points to a variable that has been deallocated

Deallocation Operator *delete*

```
int*  somePtr;  // Pointer declaration

somePtr = new int;  // Allocate a new integer variable and place its
                    // address in somePtr

delete somePtr; // Deletes the variable pointed to by somePtr, but it
                // does not delete the pointer variable itself

// delete should only be applied to a pointer value previously obtained
// from the new operator

char* someArray;
someArray = new char[6];

delete [] someArray;    // Deallocates array pointed to by someArray
```

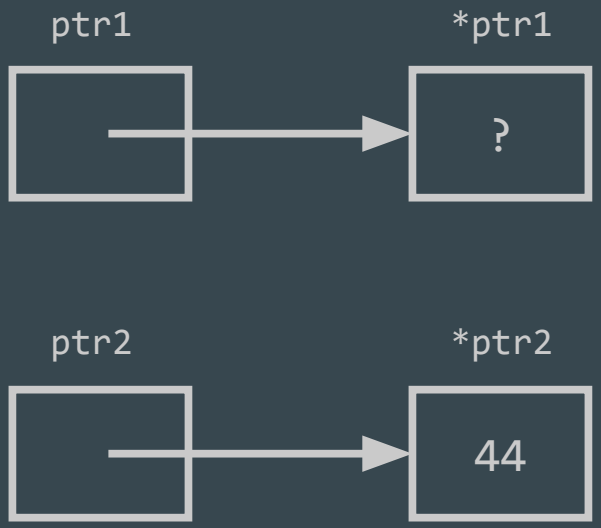
Deallocation Example

```
int* ptr1 = new int;
int* ptr2 = new int;
```



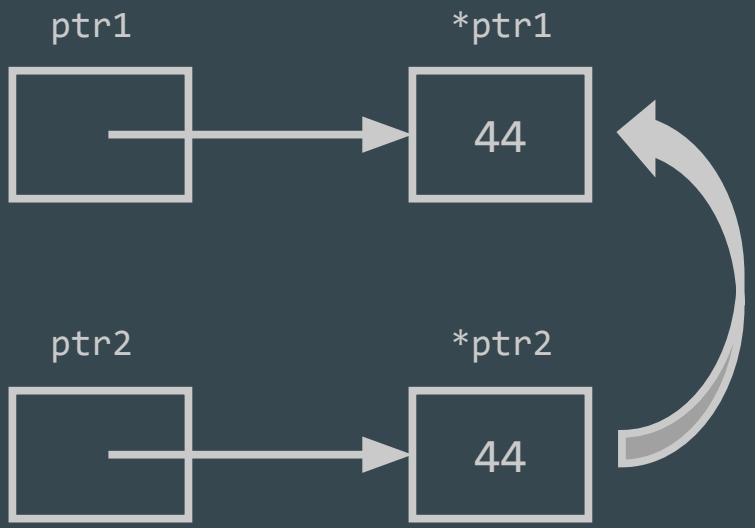
Deallocation Example

```
int* ptr1 = new int;
int* ptr2 = new int;

*ptr2 = 44;
```



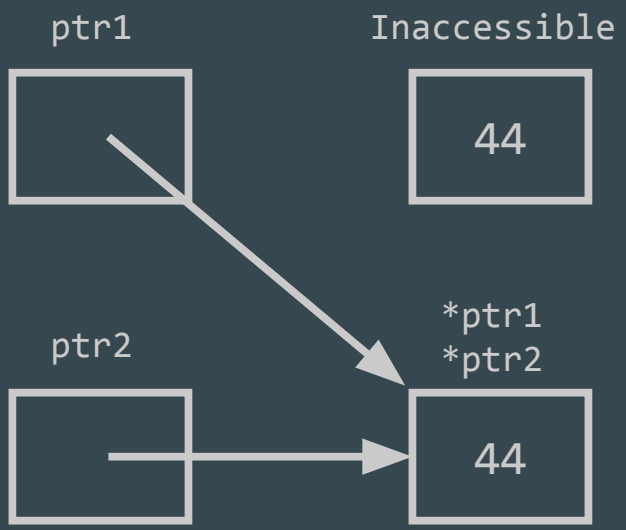
Deallocation Example

```
int* ptr1 = new int;
int* ptr2 = new int;

*ptr2 = 44;

*ptr1 = *ptr2;
```



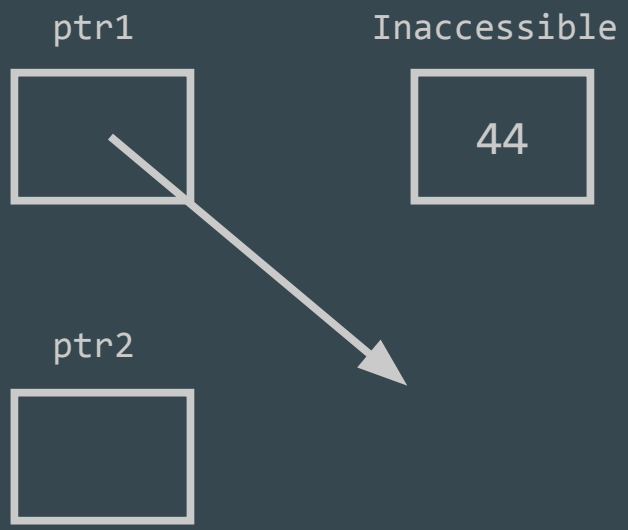
Deallocation Example

```
int* ptr1 = new int;
int* ptr2 = new int;

*ptr2 = 44;

*ptr1 = *ptr2;

ptr1 = ptr2;
```



Deallocation Example

```
int* ptr1 = new int;
int* ptr2 = new int;

*ptr2 = 44;

*ptr1 = *ptr2;

ptr1 = ptr2;

delete ptr2;   // Makes ptr1 a
            // dangling pointer

// How might one fix this code??
```



Deallocation Example

```
int* ptr1 = new int;
int* ptr2 = new int;

*ptr2 = 44;

*ptr1 = *ptr2;

delete ptr1;    // Prevent inaccessible
                // object

ptr1 = ptr2;

ptr1 =  NULL;  // Prevent dangling pointer
delete ptr2;   // Returns int variable memory
              // to heap
```