Pointers

CPE 212

UAHuntsville

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 variable declaration syntax

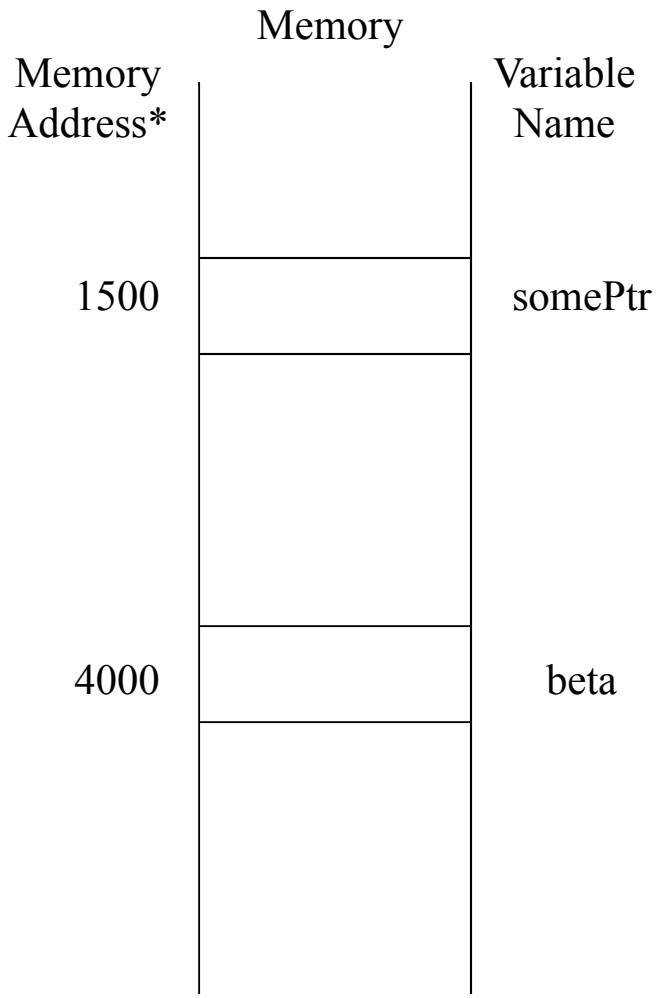
  DataType* Variable;
  or
  DataType *Variable1 [, *Variable2] … ;

Warning:

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

Pointer Examples - 1

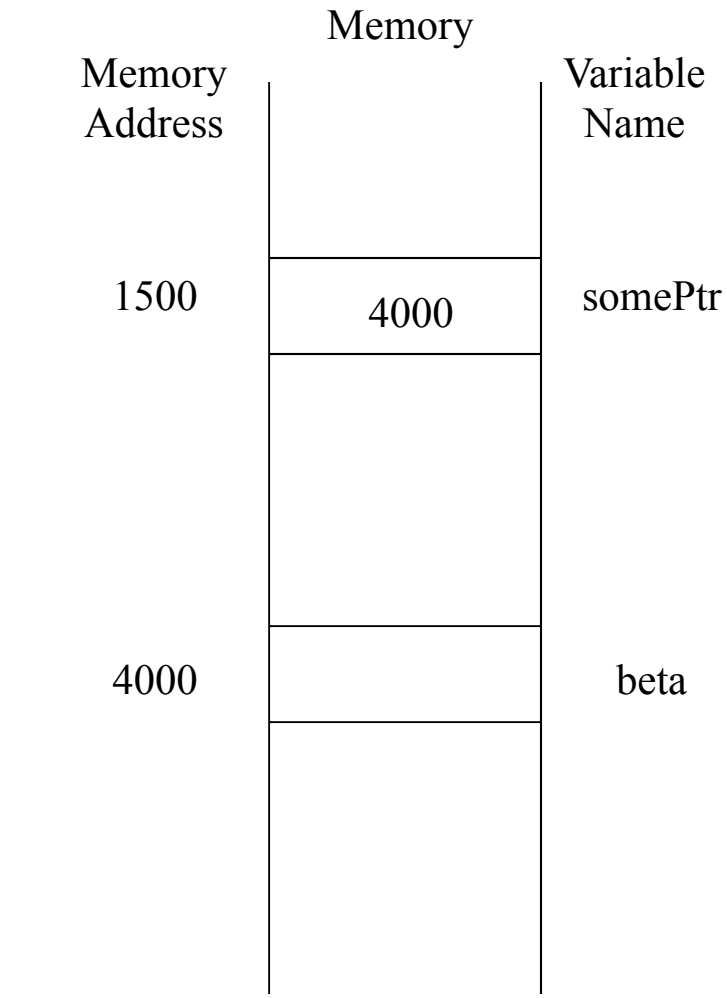
```
int   beta;      // Normal int variable
int*  somePtr;   // Pointer declaration
```



* Memory Addresses are typically written as hexadecimal values. Decimal values are used in these slides for simplicity.

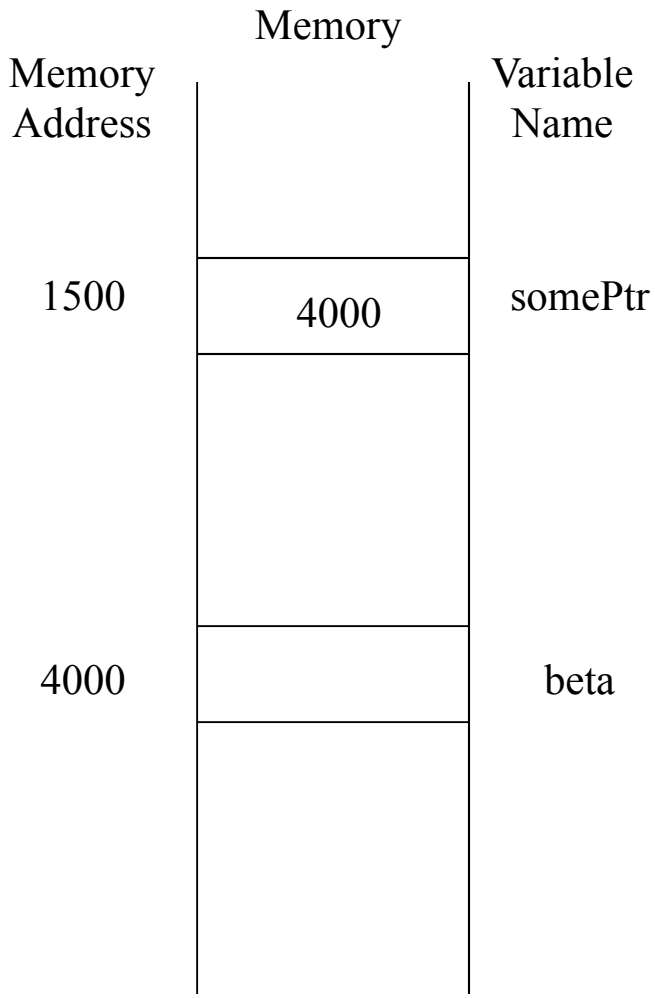
Pointer Examples - 2

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

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



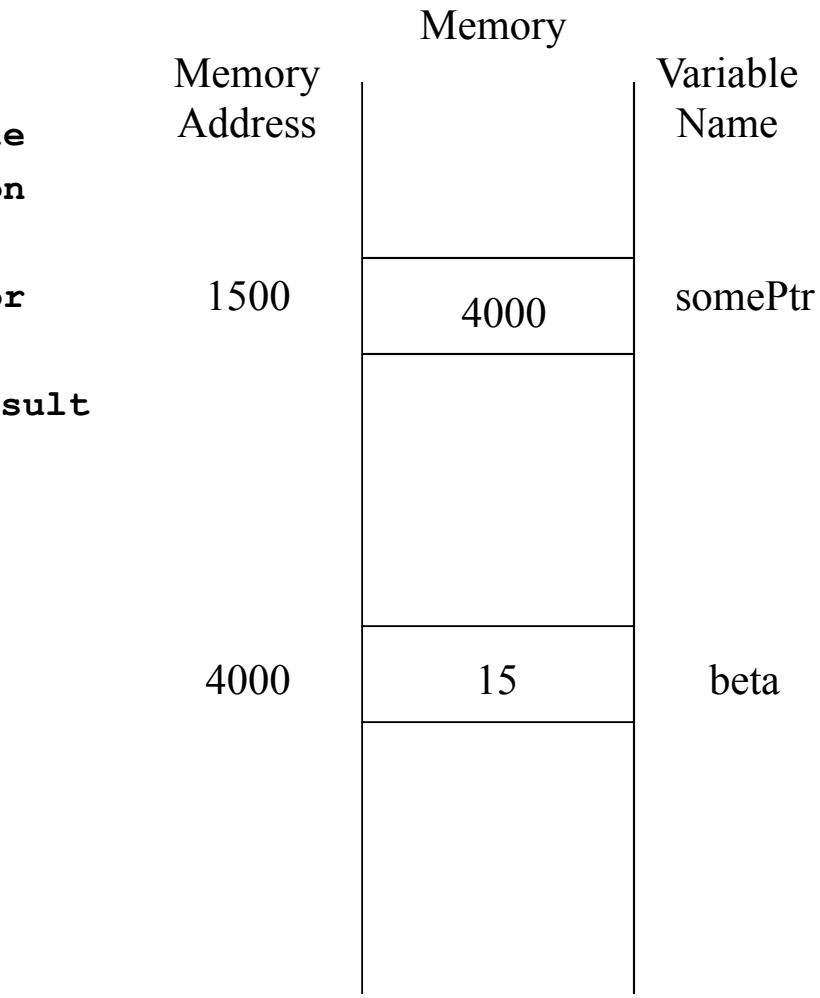
Pointer Examples - 3

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

somePtr = &beta; // Address-of operator

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



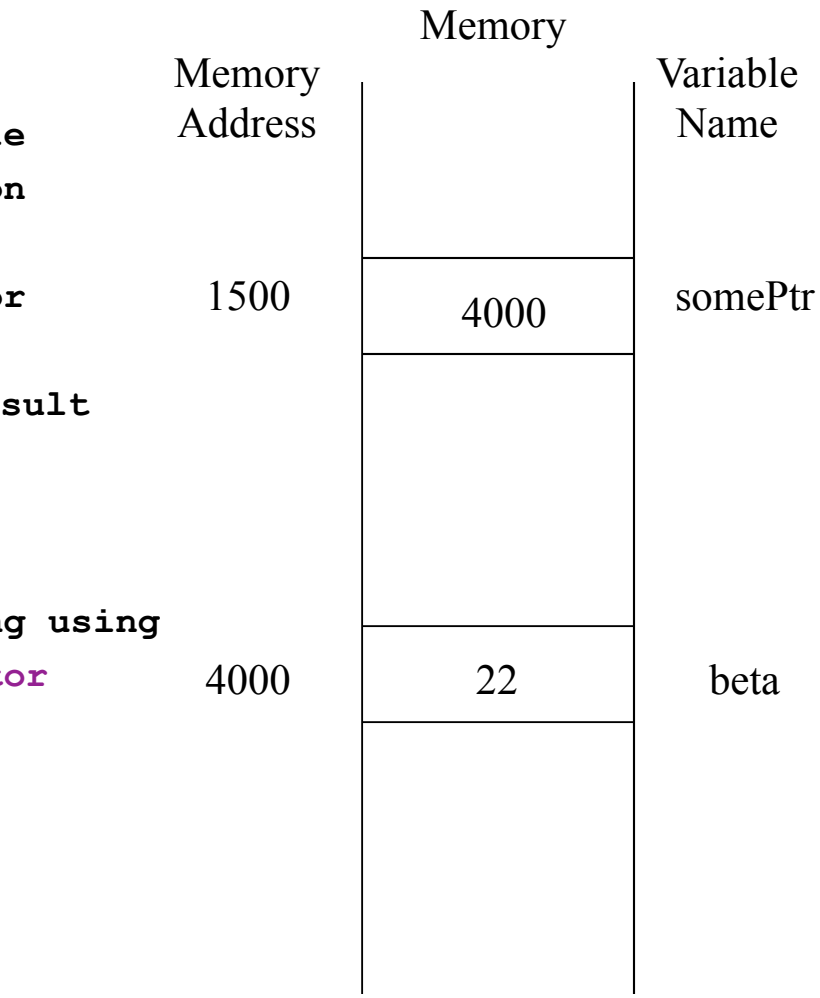
Pointer Examples - 4

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

somePtr = &beta; // Address-of operator

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

beta = 15;        // Direct addressing
```



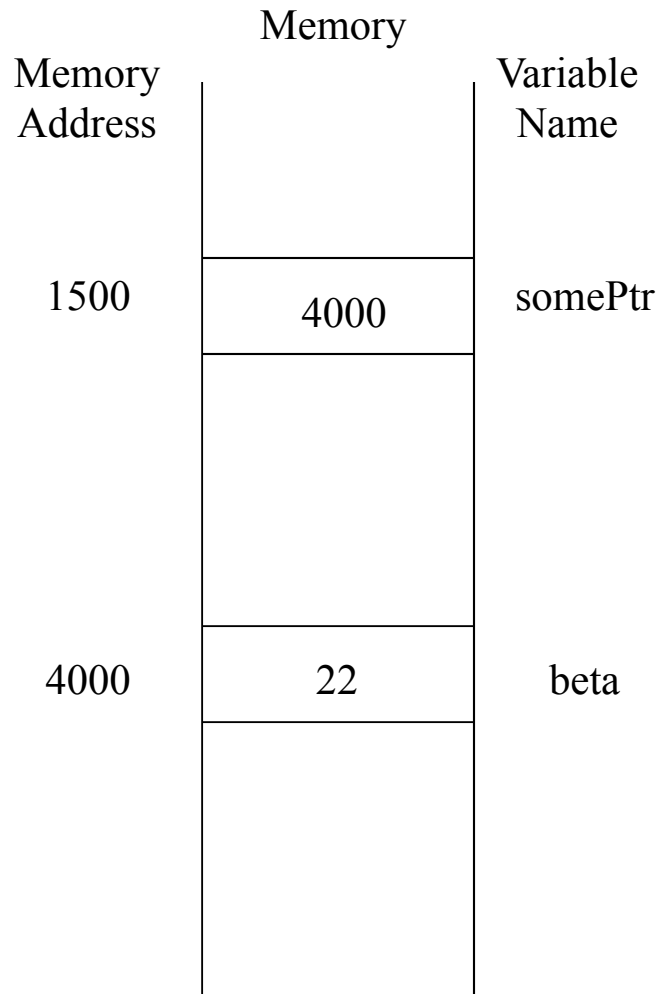
Pointer Examples - 5

```
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
```



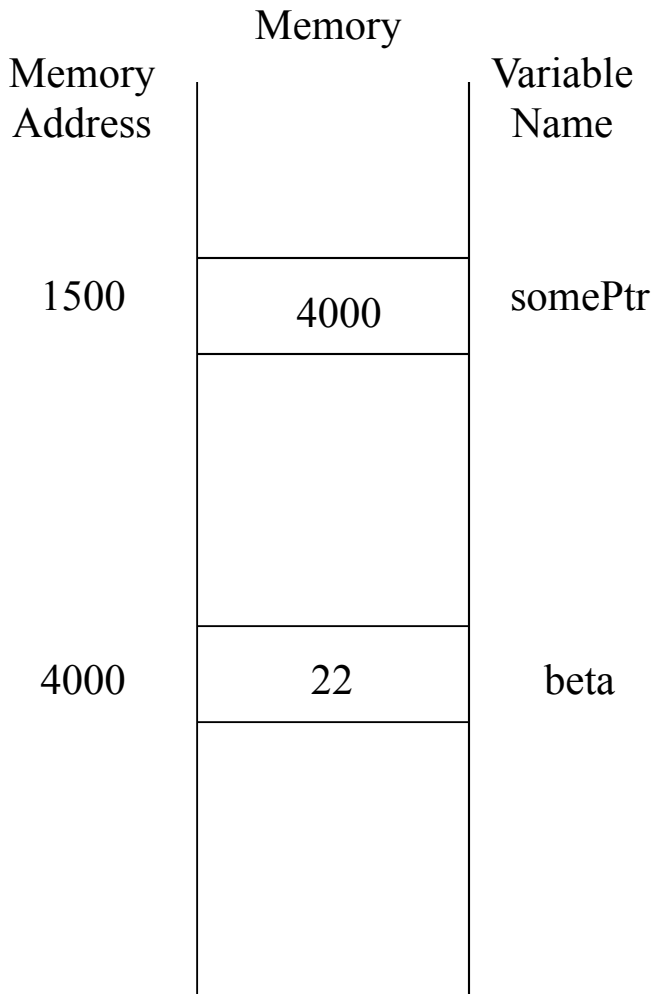
Pointer Examples - 6

```
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 *

// Pointer Questions…
cout << somePtr << endl;  // Prints what??
```



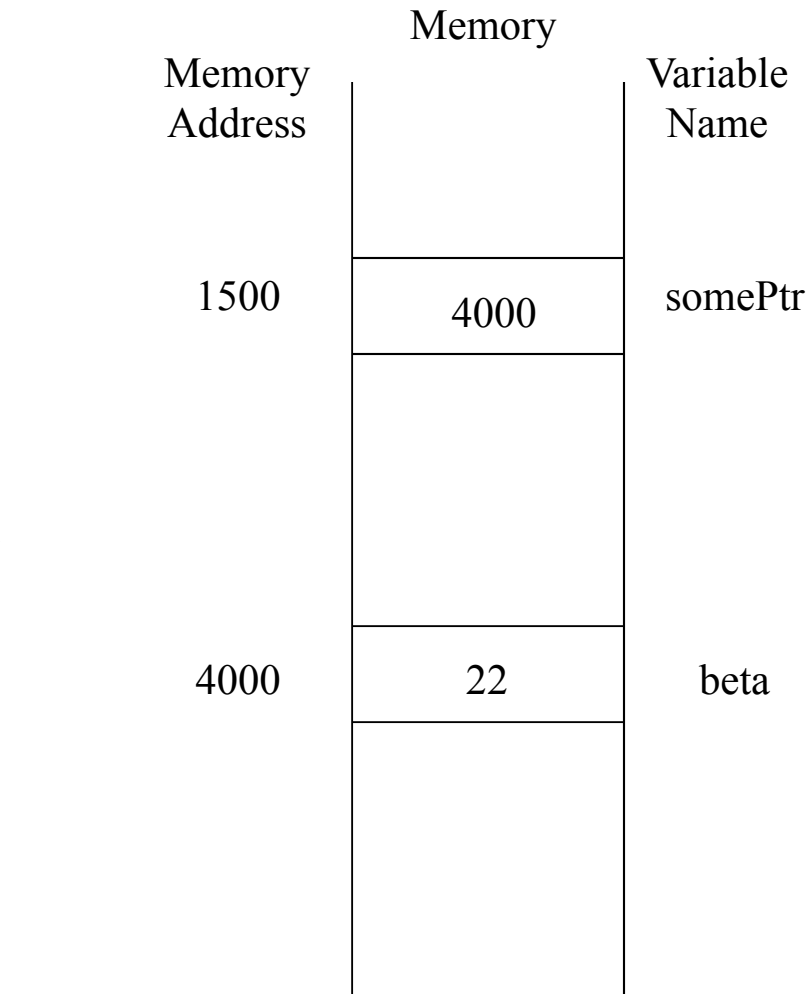
Pointer Examples - 7

```
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 *

// Pointer Questions…
cout << somePtr << endl;  // Prints 4000
cout << *somePtr << endl; // Prints what??
```



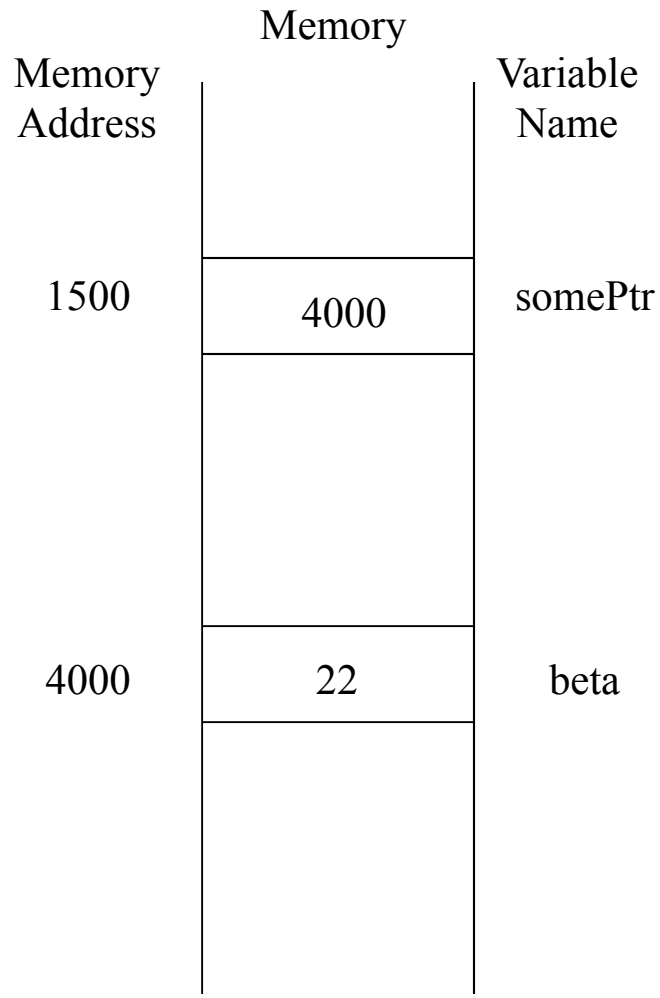
Pointer Examples - 8

```
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 *

// Pointer Questions…
cout << somePtr << endl;  // Prints 4000
cout << *somePtr << endl; // Prints 22
cout << &somePtr << endl; // Prints what??
```



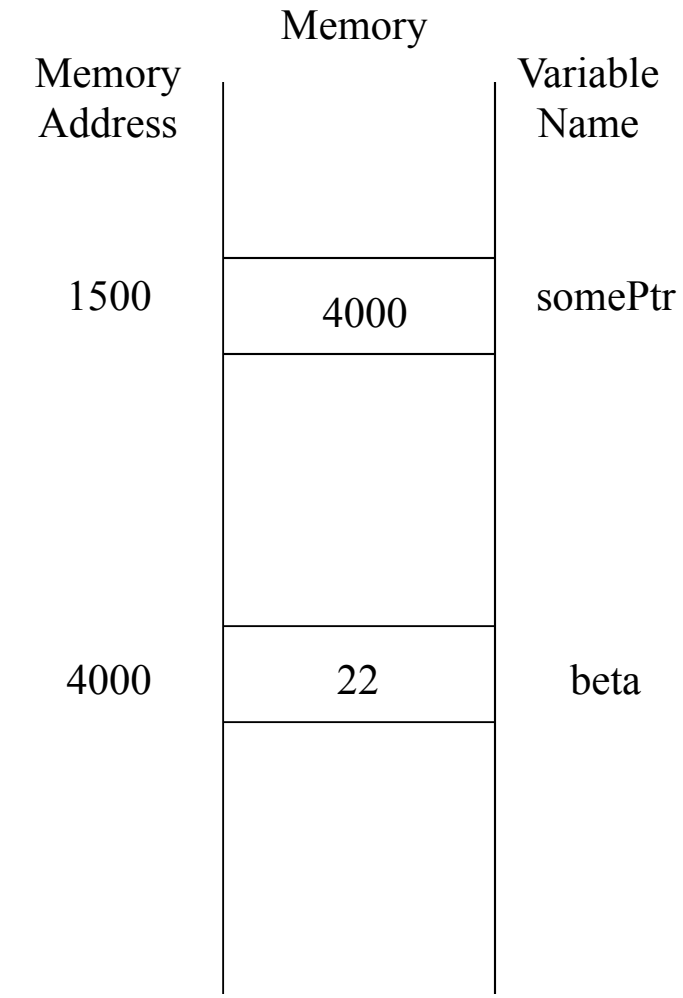
Pointer Examples - 9

```
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 *

// Pointer Questions…
cout << somePtr << endl;  // Prints 4000
cout << *somePtr << endl; // Prints 22
cout << &somePtr << endl; // Prints 1500
```



Pointer Examples - 10

```
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";
Etc.
```

More on Pointers 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?
```

More on Pointers 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***
  - Allocation of memory space at run time
- Where does the memory come from?
  - ***Free Store*** (***Heap***)
    - Pool of memory locations reserved for allocation and de-allocation of dynamic data

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
```

More Dynamic Allocation Terms

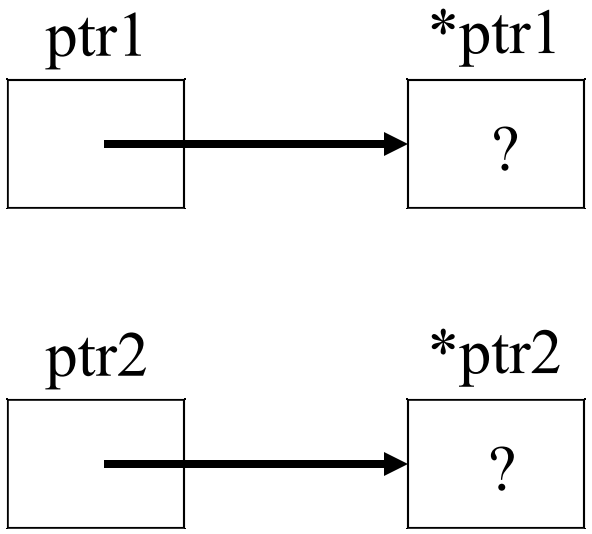
- ***Memory Leak***
  - Loss of available memory space when that memory is dynamically allocated but not deallocated
- ***Garbage***
  - Memory locations that can no longer be accessed
- ***Inaccessible Object***
  - A dynamic variable on the free store without any pointer pointing to it
- ***Dangling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

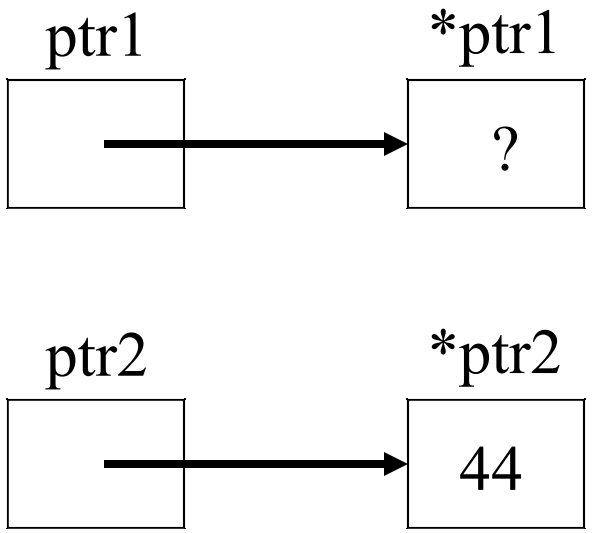
More Deallocation Examples

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



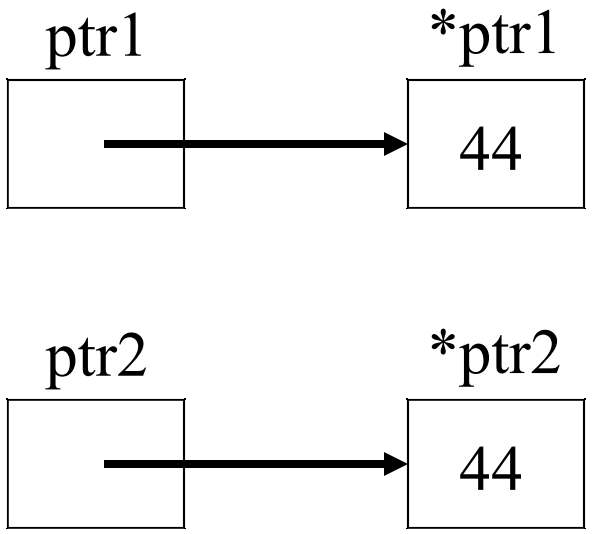
More Deallocation Examples

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

*ptr2 = 44;
```



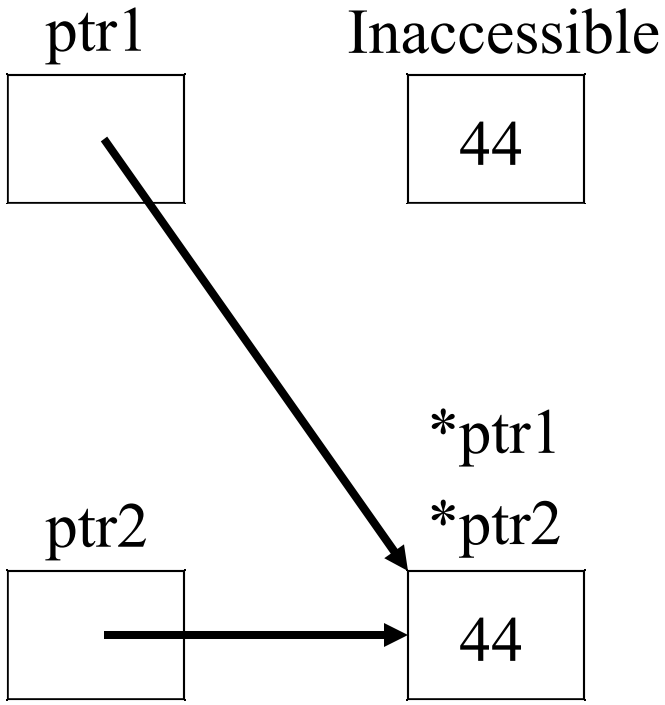
More Deallocation Examples

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

*ptr2 = 44;

*ptr1 = *ptr2;
```



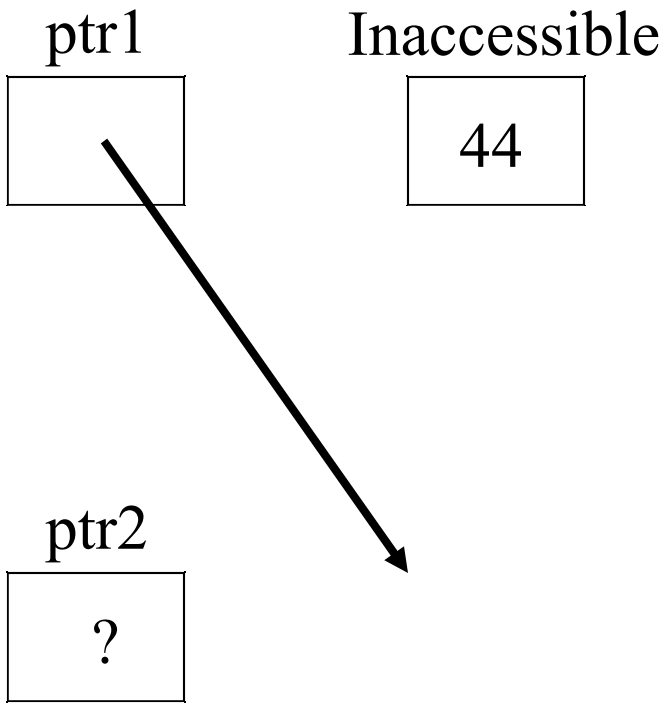
More Deallocation Examples

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

*ptr2 = 44;

*ptr1 = *ptr2;

ptr1 = ptr2;
```



More Deallocation Examples

```
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??
```



More Deallocation Examples

```
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
```