Pointers

Chapter 4

Pages 153-167

Pointer

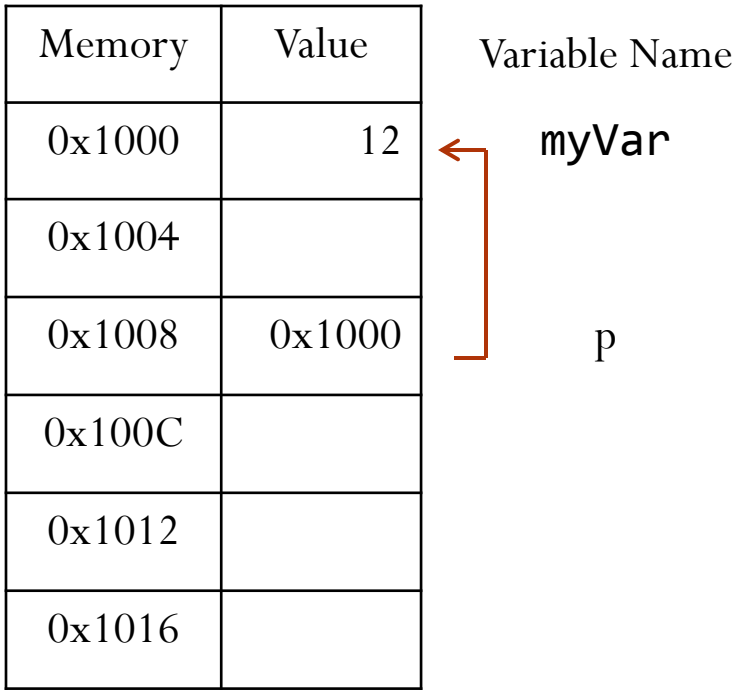
- Pointer:
  - A variable whose purpose is to store a memory address

```
int myVar = 12;
int *p;

// Fetches and Stores
// Address of Number
p = &myVar;
```

- Every variable is stored in a unique memory location

- *p* is a pointer which stores the memory location of *myVar*



Declaring a Pointer

- General Case

datatype *pointer_name;

- Specific Case

```
double *myPtr;
```

- The asterisk must go before the pointer name
  - It denotes the variable to be a pointer
- The data type of the pointer must match the datatype of the variable's address your are storing
  - i.e. if your storing an address of a float, you must use a float pointer
- You can store the address of any data type
  - even user defined structures

Address Operator

- Address Operator (&)
  - Fetches the memory address of the operand
  - Example:

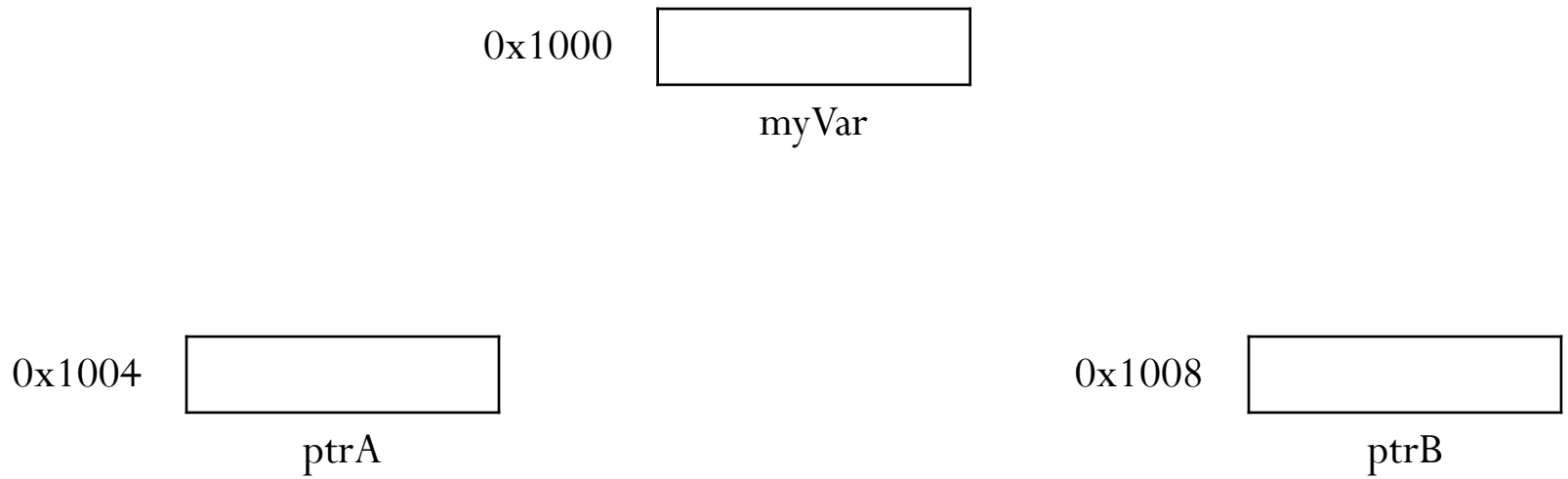
```
int myVar;
cout << &myVar << endl;
```

Prints the memory address of *myVar*

- The (&) operator can have entirely different meanings depending on how it is used:
  - In Declaration: aliasing
    - int &a = b;
  - Applied to a single operand: address operator
    - &myVar
  - Applied to two operands: bitwise And
    - a & b

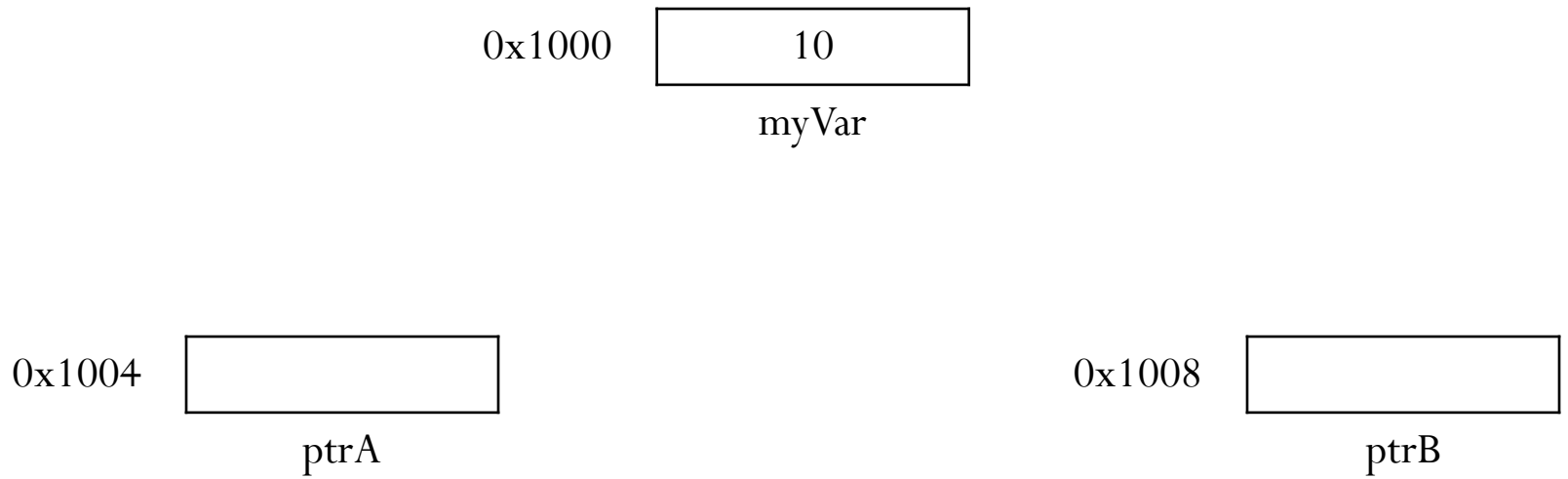
Storing an Address

```
int myVar = 10;
int *ptrA, *ptrB;
ptrA = &myVar;
ptrB = ptrA;
```



Example of Using Address Operator

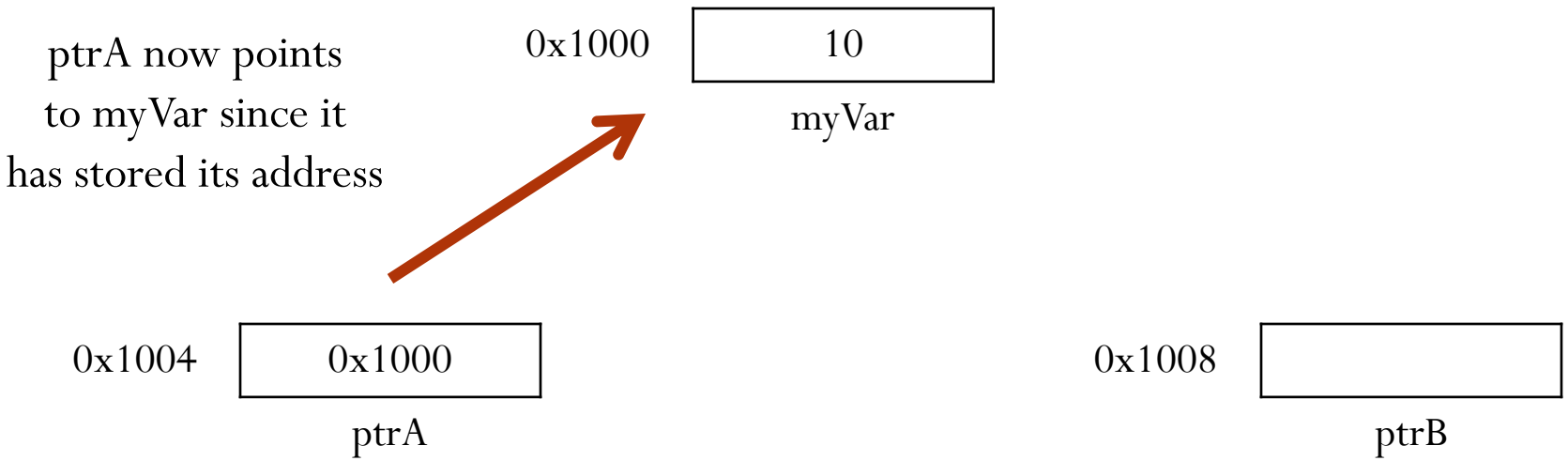
```
int myVar = 10;
int *ptrA, *ptrB;
ptrA = &myVar;
ptrB = ptrA;
```



Example of Using Address Operator

```
int myVar = 10;
int *ptrA, *ptrB;
ptrA = &myVar;
ptrB = ptrA;
```

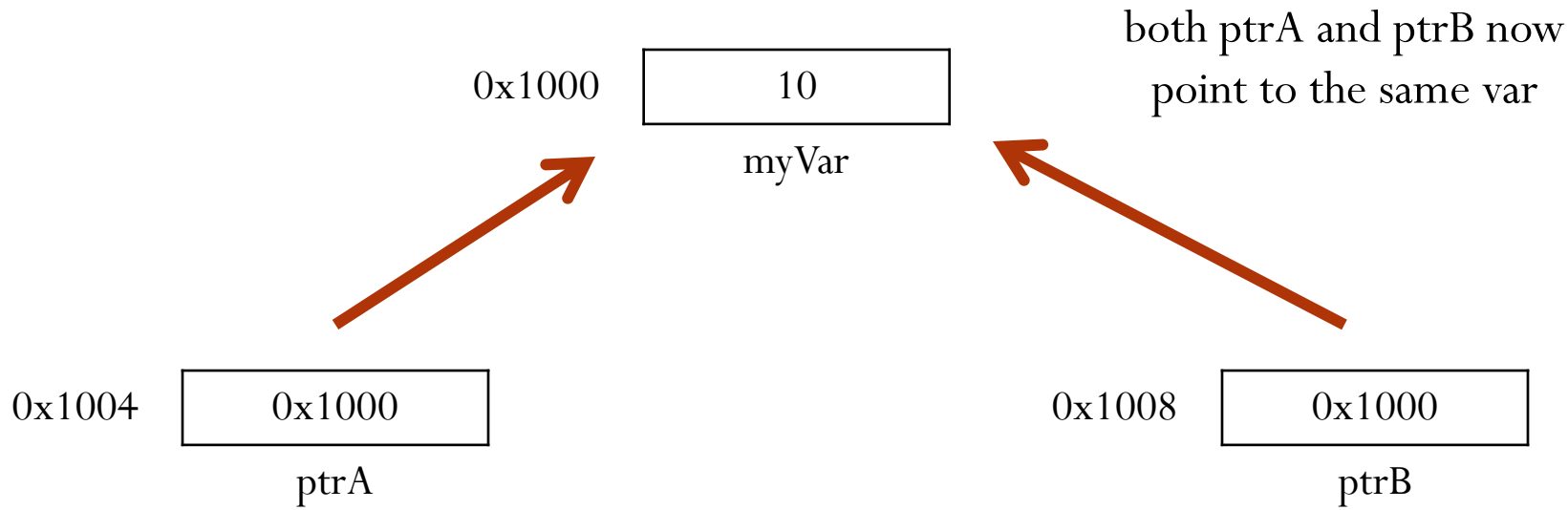
- Here the memory location of myVar is fetched using the address operator
- The value is then stored into a pointer



Example of Using Address Operator

```
int myVar = 10;
int *ptrA, *ptrB;
ptrA = &myVar;
ptrB = ptrA;
```

- The value contained in a pointer can be copied just like any other variable



Deference Operator

- Deference Operator (*)
  - Returns the value at the pointer address
  - Example:

```
int myVar = 10;
int *ptrA;
ptrA = &myVar;
cout << *ptrA << endl;
```

Prints the contents of myVar, which is 10

Dereference Operator

- The (*) operator can have entirely different meanings depending on how it is used:
  - In Declaration: pointer variable
    - int *a = b;
  - Applied to a single operand: deference operator
    - *ptrA
  - Applied to two operands: multiplication
    - a * b

- This can be confusing, but we will review of few examples

Dereference Operator

- The (*) operator can have entirely different meanings depending on how it is used:
  - In Declaration: pointer variable
    - int *a = b;
  - Applied to a single operand: deference operator
    - *ptrA
  - Applied to two operands: multiplication
    - a * b

- This can be confusing, but we will review of few examples

Dereference Example

```
int myVar = 10;
int *ptrA;
ptrA = &myVar;
cout << *ptrA << endl;
```

Dereference Example

```
int myVar = 10;
int *ptrA;
ptrA = &myVar;
cout << *ptrA << endl;
```

- Assign 10 into myVar

Dereference Example

```
int myVar = 10;
int *ptrA;
ptrA = &myVar;
cout << *ptrA << endl;
```

- Store address of myVar into ptrA
- ptrA now points to myVar

Dereference Example

```
int myVar = 10;
int *ptrA;
ptrA = &myVar;
cout << *ptrA << endl;
```

- Dereference called on ptrA
  - Fetch value at the location ptrA points to
- Prints 10 to the console

Example 14.1

```
int b ;
int *a = &b;
*a = 10;
cout << b << endl;
```

- Here we use the dereference to store the value 10 in b

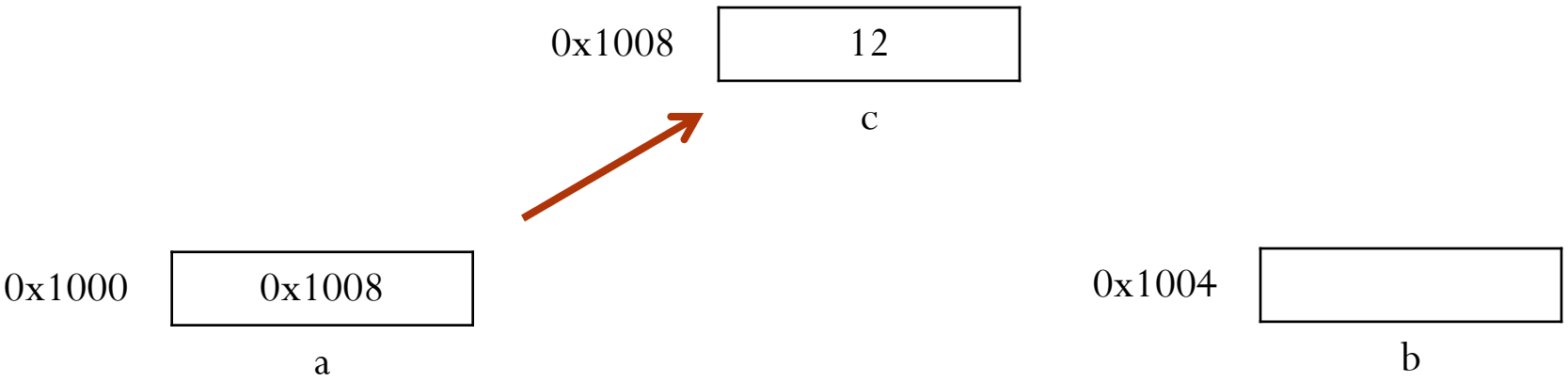
Step 1: int *a = &b;

Step 2: *a = 10;

Example 14.2

```
int *a, *b;
int c = 12;
a = &c;
b = a;
*b = 15;
```

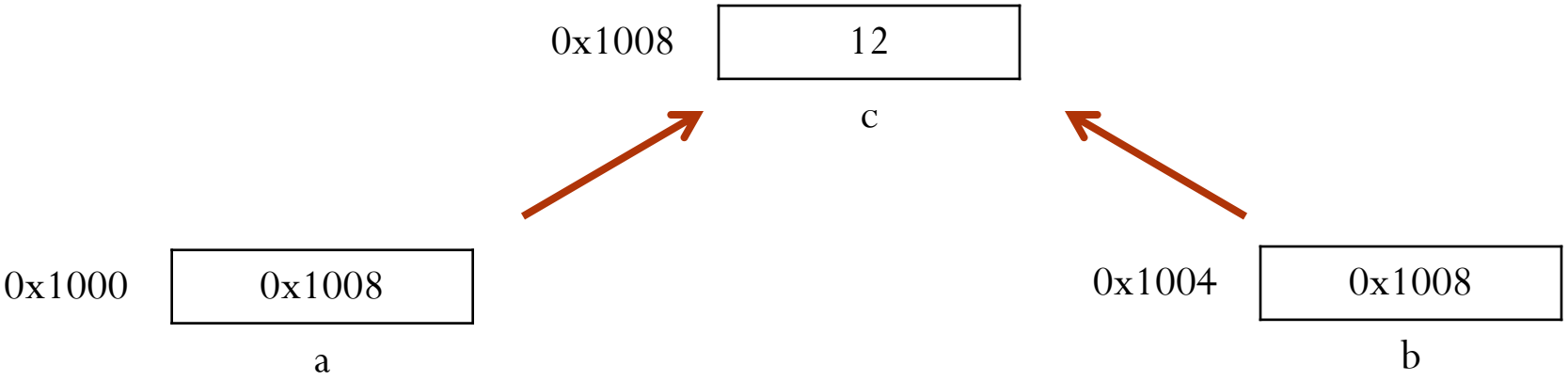
- Store 12 into C
- Store address of C into a



Example 14.2

```
int *a, *b;
int c = 12;
a = &c;
b = a;
*b = 15;
```

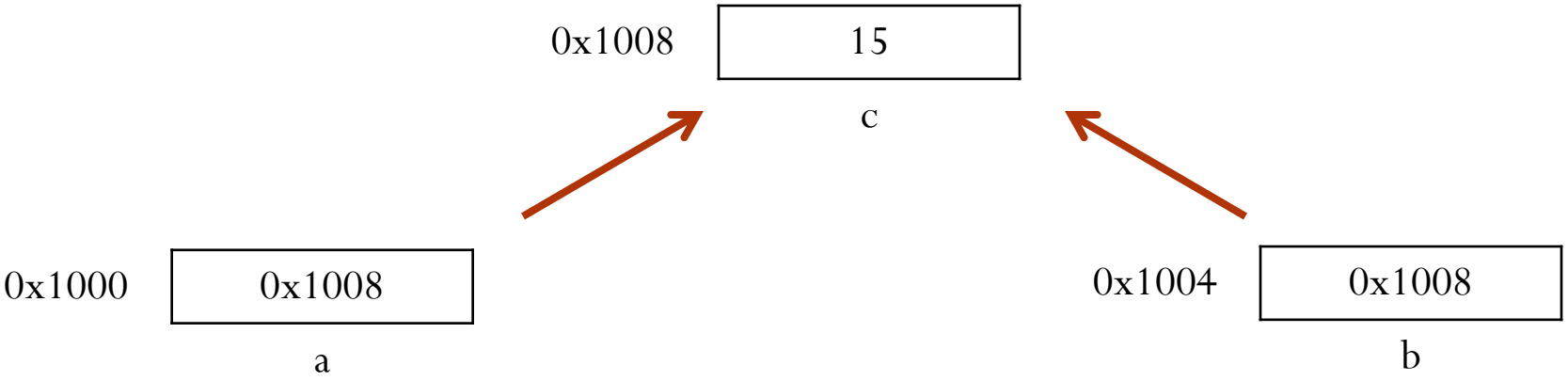
- Copy contents of a into b



Example 14.2

```
int *a, *b;
int c = 12;
a = &c;
b = a;
*b = 15;
```

- Store 15 at the location pointer b points to



Example 14.3

```
int a = 12;
int *b;
b = a;
*b = *b + 1;
```

- What is wrong with the code to the left?
- How can it be corrected?

- b = a
- b cannot store a integer value, only an address of an integer variable
- therefore b cannot store 12
- Corrected code:

```
int a = 12;
int *b;
b = &a;
*b = *b + 1;
```

Example 14.4

```
int a = 12;
int *b;
*b = &a;
*b = *b + 1;
```

- What is wrong with the code to the left?
- How can it be corrected?

- *b = &a
- Here we are trying to store the address of *a* into the location to which b points at
- There are two problems with this:
  - pointer b is uninitialized (likely will crash the program)
  - a dereference of a integer pointer will be expecting an integer, not a memory address
- Corrected code:

```
int a = 12;
int *b;
b = &a;
*b = *b + 1;
```

Arrays and Pointers

- An array is stored as consecutive addresses as shown to the right

```
int x[4] = {2, 4, 8, 10};
```

- What happens when we cout x?

cout << x << endl;

- The address of x[0] is printed (0x1000)

Memory	Value	Variable Name
0x1000	2	x[0]
0x1004	4	x[1]
0x1008	8	x[2]
0x100C	10	x[3]
0x1012		
0x1016		

Access Array through Pointer Notation

- Using the previous fact what will the following print?

```
cout << *x << endl;
```

- x[0] will be printed, 2

- What about the following?

```
cout << *(x+1) << endl;
```

Memory	Value	Variable Name
0x1000	2	x[0]
0x1004	4	x[1]
0x1008	8	x[2]
0x100C	10	x[3]
0x1012		
0x1016		

Access Array through Pointer Notation

```
cout << *(x+1) << endl;
```

- (x+1) adds one integer's worth of bytes to the address of x
- Thus (x+1) evaluates to 0x1004
- *(0x1004) will return 4
- Therefore 4 will be printed
- Equivalent to x[1]

Memory	Value	Variable Name
0x1000	2	x[0]
0x1004	4	x[1]
0x1008	8	x[2]
0x100C	10	x[3]
0x1012		
0x1016		

Access Array through Pointer Notation

- What about?

```
cout << *(x+2) << endl;
```

- Or

```
cout << *(x+3) << endl;
```

- 8 and 10 will be printed respectively

Memory	Value	Variable Name
0x1000	2	x[0]
0x1004	4	x[1]
0x1008	8	x[2]
0x100C	10	x[3]
0x1012		
0x1016		

Access Array through Pointer Notation

- What about?

```
cout << *x + 1 << endl;
```

- 3 will be printed
- The above code first dereferences at x yielding 2
- Thus

cout << 2 + 1 << endl;

Memory	Value	Variable Name
0x1000	2	x[0]
0x1004	4	x[1]
0x1008	8	x[2]
0x100C	10	x[3]
0x1012		
0x1016		

Print arrays through address notation

- Can we write a loop to print this array?

```
int x[4] = {2, 4, 8, 10};
int i;

for(i = 0; i < 4; i++)
    cout << *(x+i) << endl;
```

- The compiled assembly language closely reflects the above operation which you use x[i] notation

Memory	Value	Variable Name
0x1000	2	x[0]
0x1004	4	x[1]
0x1008	8	x[2]
0x100C	10	x[3]
0x1012		
0x1016		

What's the point of pointers?

- Clearly pointers are fascinating but I already have ways to access arrays, and to store information into variables

- Pointers allow for some non-obvious powerful operations which include the following:
  - Dynamic Memory
  - Pass by Reference
  - Function Pointers
  - Aliasing
  - Linked Lists