CSC 211: Object Oriented Programming

Pointers

Michael Conti

Department of Computer Science and Statistics
University of Rhode Island

Spring 2020

THINK BIG WE DO℠

Original design and development by Dr. Marco Alvarez

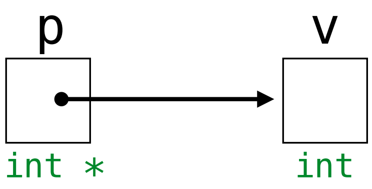
So far …

- Every variable/object (regardless of scope) exists at some memory location (**memory address**)
- Every memory address corresponds to a **unique location** in memory
- The compiler translates names into memory addresses when generating machine level code
- C++ allows programmers to manipulate variables/objects and their memory addresses directly

What is a pointer?

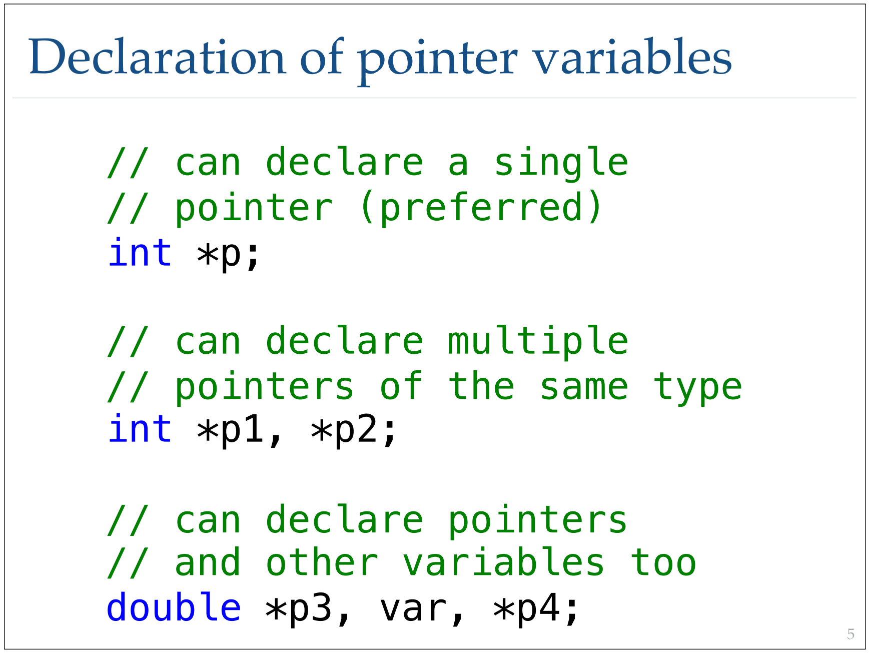
- A special type of variable whose value is the **memory address** of another variable
- Pointers must be **declared** before use
  - pointer type **must** be specified
  - pointers **must always** point to variables/objects of the same type

A pointer **p** that stores the memory address of another variable **v** is said to **point** to **v**



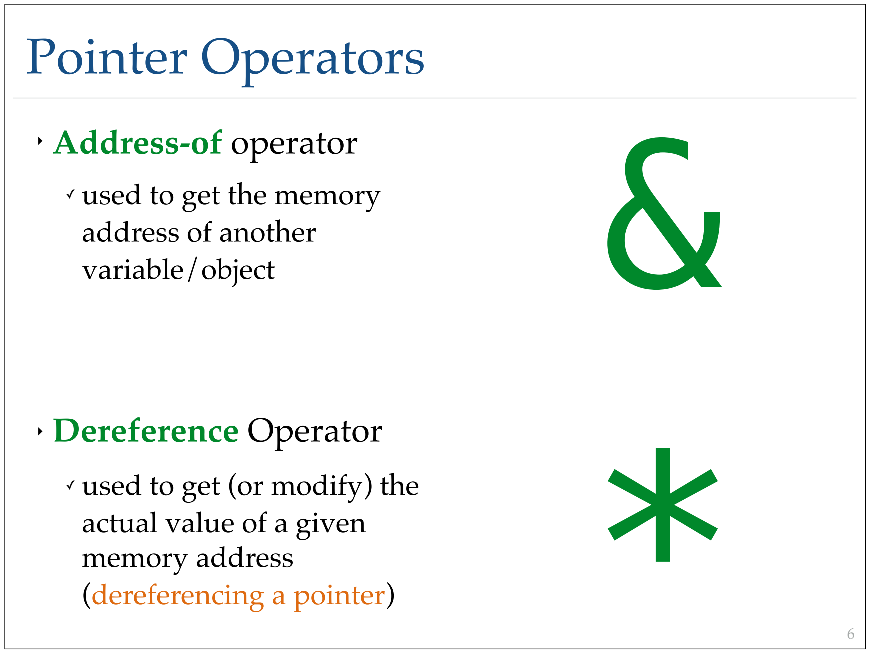
Declaration of pointer variables

```
type *ptr_name;
```



Declaration of pointer variables

```
// can declare a single
// pointer (preferred)
int *p;

// can declare multiple
// pointers of the same type
int *p1, *p2;

// can declare pointers
// and other variables too
double *p3, var, *p4;
```



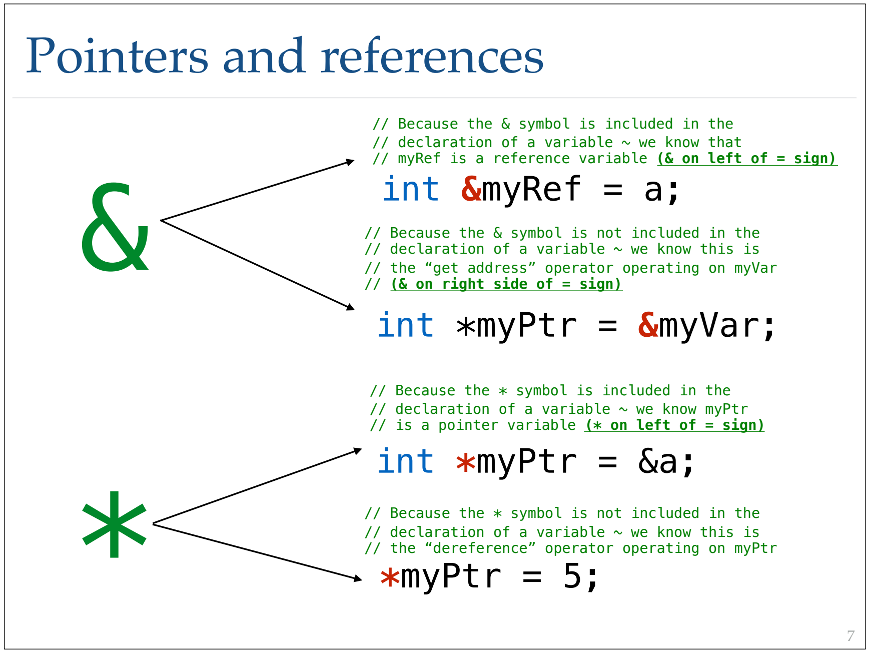
Pointer Operators

- **Address-of** operator
  - used to get the memory address of another variable / object

&

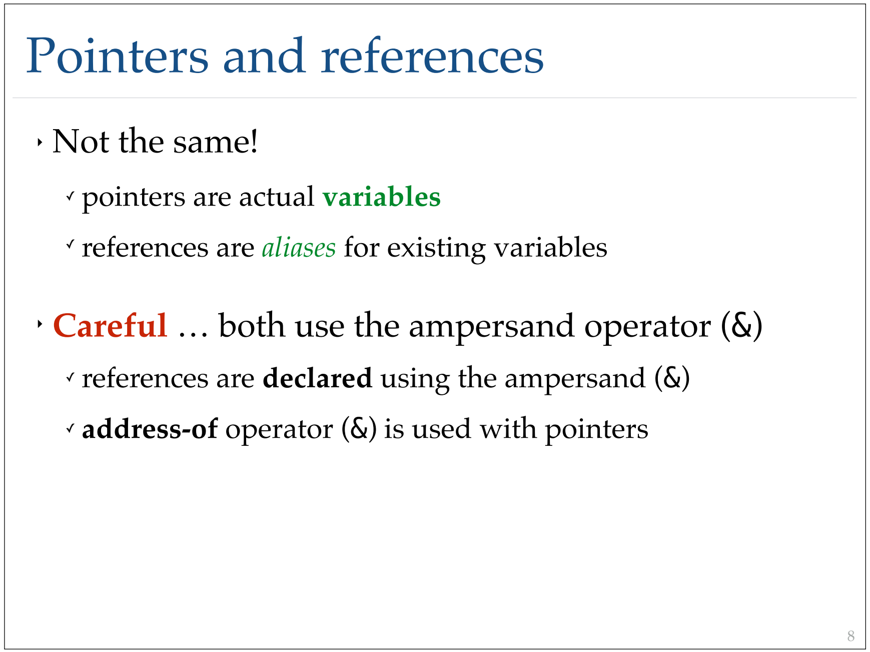
- **Dereference** Operator
  - used to get (or modify) the actual value of a given memory address (dereferencing a pointer)

*



Pointers and references

```
// Because the & symbol is included in the
// declaration of a variable ~ we know that
// myRef is a reference variable (& on left of = sign)
int &myRef = a;

// Because the & symbol is not included in the
// declaration of a variable ~ we know this is
// the “get address” operator operating on myVar
// (& on right side of = sign)
int *myPtr = &myVar;

// Because the * symbol is included in the
// declaration of a variable ~ we know myPtr
// is a pointer variable (* on left of = sign)
int *myPtr = &a;

// Because the * symbol is not included in the
// declaration of a variable ~ we know this is
// the “dereference” operator operating on myPtr
*myPtr = 5;
```



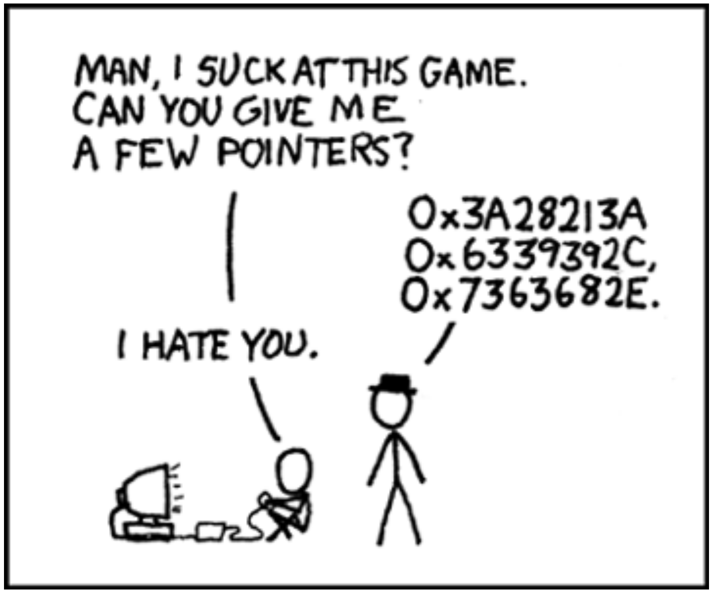
Pointers and references

- Not the same!
  - pointers are actual **variables**
  - references are *aliases* for existing variables
- **Careful** … both use the ampersand operator (&)
  - references are **declared** using the ampersand (&)
  - **address-of** operator (&) is used with pointers

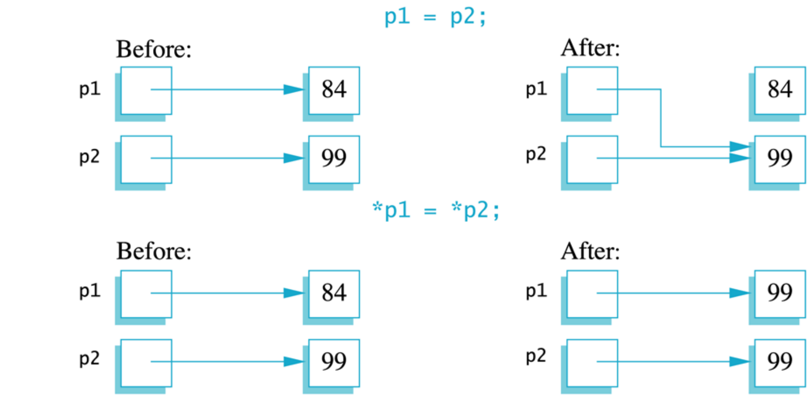
```
#include <iostream>

int main() {
    int var = 10;
    int *ptr;

    ptr = &var;
    *ptr = 20;

    // print both
    // using cout
    cout << var;
    cout << ptr;

    cout << *ptr;
    return 0;
}
```

Assuming 32-bit words

Address	Variable	Value
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		
0x91340A24		
0x91340A28		
0x91340A2C		
0x91340A30		
0x91340A34		
…		



Uses of the Assignment Operator



```
int main() {
    int temp = 10;
    int value = 100;
    int *p1, *p2;

    p1 = &temp;
    *p1 += 10;

    p2 = &value;
    *p2 += 5;

    p2 = p1;
    *p2 += 5;

    return 0;
}
```

Address	Variable	Value
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		

Null pointers and functions

- Pointers can be initialized to an "empty" address (points to nothing) using the **nullptr** keyword
    - **nullptr** is just a pointer literal
- Pointers can be passed as parameters to functions
    - pointers are **treated as any other variable**
    - just remember they are holding **memory addresses**

```
#include <iostream>

void increment(int *ptr) {
    (*ptr) ++;
}

int main() {
    int var = 10;

    increment(&var);
    increment(&var);

    // print using cout

    return 0;
}
```

Address	Variable	Value
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		
0x91340A24		
0x91340A28		
0x91340A2C		
0x91340A30		
0x91340A34		
…		

Pointers and arrays

- When declaring an array, the array name is treated as a **constant pointer** (pointing to the **base address**)

```
void zeros(int a[], int n){
    for (int i = 0 ; i < n ; i ++){
        a[i] = 0;
    }
}

int main() {
    int array[5];
    zeros(array, 5);
    // do stuff
}
```

=

```
void zeros(int *a, int n) {
    for (int i = 0 ; i < n ; i ++){
        a[i] = 0;
    }
}

int main() {
    int array[5];
    zeros(array, 5);
    // do stuff
}
```

Pointer arithmetic

- As pointers hold **memory addresses** (basically integers), we can add integers to it
- Must be careful !
    - `p+1` does not add 1 byte to the memory address, it adds the **size of the variable pointed by `p`**

```
int *myPtr = &a;
myPtr is holding 0x7ffee7e44bcc
myPtr + 1 == 0x7ffee7e44bcc + 1 =
0x7ffee7e44bd0 (4 bytes were added)
```

- Can use pointer arithmetic to work with arrays

Example

- Implement `reverse a string` using pointer arithmetic