ACA Summer School 2014
Advanced C++

Pankaj Prateek

ACA, CSE, IIT Kanpur

June 26, 2014

Pointer: Basics

- Variable

  Name/Identifier $\leftrightarrow$ Contains a value $\leftrightarrow$ Memory address

- Memory cells are numbered in continuation, giving every cell a unique number, which is called its address
- Pointer

  Name/Identifier $\leftrightarrow$ Contains memory address of another variable

- Pointers contain the address of some other variable. They are said to "point to" that variable

Pointer: Basics

- Variable

  Name/Identifier $\leftrightarrow$ Contains a value $\leftrightarrow$ Memory address
- Memory cells are numbered in continuation, giving every cell a unique number, which is called its address
- Pointer

  Name/Identifier $\leftrightarrow$ Contains memory address of another variable
- Pointers contain the address of some other variable. They are said to "point to" that variable

Pointer: Basics

- Variable

  Name/Identifier $\leftrightarrow$ Contains a value $\leftrightarrow$ Memory address
- Memory cells are numbered in continuation, giving every cell a unique number, which is called its address
- Pointer

  Name/Identifier $\leftrightarrow$ Contains memory address of another variable
- Pointers contain the address of some other variable. They are said to "point to" that variable

Pointer: Basics

- Variable

  Name/Identifier $\leftrightarrow$ Contains a value $\leftrightarrow$ Memory address
- Memory cells are numbered in continuation, giving every cell a unique number, which is called its address
- Pointer

  Name/Identifier $\leftrightarrow$ Contains memory address of another variable
- Pointers contain the address of some other variable. They are said to "point to" that variable

Pointer: Basics

```
int a;
int *ptr = &a; //Reference operator
a = 10;
*ptr = 12;    //Dereference operator
```

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointer: Basics

- Pointers are type-specific (that is there is a different pointer for every different type of variable)
- Pointer to a pointer is also possible (double reference)
- Extremely useful in call by reference mechanism
- Also useful in dynamic memory allocation
- Void pointer : a type-less pointer, but cannot be dereferenced without explicit type cast
- Null pointer : A pointer that points to nothing or 0. Cannot be derefrenced. Doing that raises a run-time error/segmentation fault
- Equivalent array names

Pointers to structures

```
struct temp {
  int a; float b;
};
int main() {
  struct temp obj1;
  struct temp * ptr;
  obj1.a = 10;
  obj1.b = 3.14;
  ptr = &obj1;
  cout << ptr -> a << endl;
  cout << (*ptr).b << endl;
}
```

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer Arithmetic

- ++ and −− operators are allowed on pointers
- They correspond to advancing and retreating the pointer by the size of one object in memory
- + and - are also allowed on pointers, which advance or retreat them suitably
- *(a + 10) is equivalent to a[10] while (a + 10) is equivalent to &a[10]
- Should be used with caution. Can possibly be the result of array out of bounds errors, resulting in segmentation faults
- Array names are treated as constant pointers. Operations like A = A+1 are invalid for them

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointer: Basics

- References are alias (second name) to variables
- They do not contain memory address like pointers do
- Unlike pointers, once assigned, then cannot be changed later
- There are no Null references. They have to be initialized when they are declared.
- The actual pass by reference is implemented through them, passing pointers is still only pass by value

Pointers to Objects

```
class ABC {
  int var1;
public:
  setVar(int a);
  int getVar();
};
ABC obj1;
ABC* ptr1;
ptr1 = &obj1;
ptr1->setVar(20);
cout<<ptr1->getVar();
```