C++: Pointers and Structs

Spring 2018

Topics

- Syntax
- Pointer operators
- Pass by reference using pointers
- Different usage of asterisk (*)
- Pointer arithmetic
- Struct

What is a pointer?

A pointer contains the *memory address* of a variable that, in turn, contains a specific value.

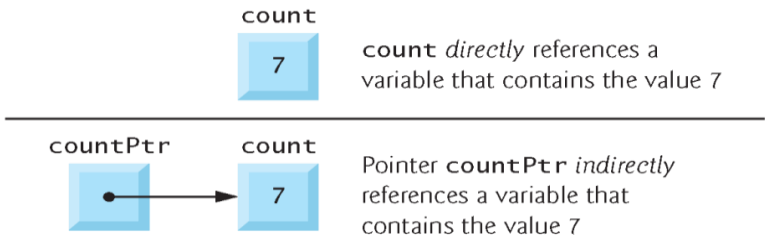


Fig. 8.1 | Directly and indirectly referencing a variable.

Pointer Declaration

- Syntax:

  <data-type> * <pointer-variable-name> ;

- When * appears in a declaration, it is *not* an operator; rather, it indicates that the variable being declared is a pointer.

- Pointers can be declared to point to objects of *any* data type.

- Pointers should be initialized to `nullptr` (new in C++11) or an address of the corresponding type either when they're declared or in an assignment.

Pointer Operator: Address Operator (&)

- The address operator (`&`) is a unary operator that *obtains the memory address of its operand*.

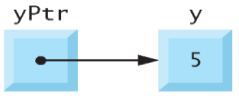


Fig. 8.2 | Graphical representation of a pointer pointing to a variable in memory.

Fig. 8.3 | Representation of `y` and `yPtr` in memory.

- For example,

```
int y = 5;
int * yPtr = &y;
```

Pointer Operator: Indirection/Dereferencing Operator (*)

- The unary * operator—commonly referred to as the indirection operator or dereferencing operator—*returns an lvalue representing the object to which its pointer operand points.*
  - Called dereferencing a pointer

- A *dereferenced pointer* may also be used on the *left* side of an assignment.

Operator Precedence Order

Operators	Associativity	Type
`::` `()`	left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]*	primary
`()` `[]` `++` `--` `static_cast<`*type*`>(`*operand*`)`	left to right	postfix
`++` `--` `+` `-` `!` `&` `*`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional

Fig. 8.5 | Operator precedence and associativity of the operators discussed so far. (Part 1 of 2.)

http://en.cppreference.com/w/cpp/language/operator_precedence

Pass-by-reference using pointers

- There are three ways in C++ to pass arguments to a function—pass-by-value, pass-by-reference with reference arguments and pass-by-reference with pointer arguments.

- Pointers, like references, can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value.

- You can use pointers and the indirection operator (*) to accomplish pass-by-reference.

- When calling a function with an argument that should be modified, the *address* of the argument is passed.

Three different uses of the asterisk (*) in C++

1. **As the multiplication operator (binary operator), in statements such as**

   total = price * quantity;

2. **In the definition of a pointer variable for declaring pointer variable (not an operator, just a C++ declaration of pointer data type)**

   int *ptr = nullptr;

3. **As the indirection operator (unary operator), in statements such as**

   ***ptr = 100;**

Pass-by-value argument

```
int doubleByValue(int val) ;          // prototype
int main ()
{
  int number = 10;
  doubleByValue(number) ;
  return 0;
}

int doubleByValue(int val)
 {
  val = val*2;
  return val;
 }
```

1. What's the value of number after calling doubleByValue()?
2. What changes would you make (without using pointer) to get number = 20?

Pass-by-reference with a pointer argument

```
void doubleByReference(int *valPtr) ;            // prototype

int main ()
{
  int number = 10;
  doubleByReference(&number) ;
}

void doubleByReference(int *valPtr)
 {
  *valPtr = (*valPtr) * 2;      // Is it confusing ?
 }
```

What's the value of number after calling doubleByReference()?

Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- C++ enables pointer arithmetic—a few arithmetic operations may be performed on pointers:
  - increment (++)
  - decremented (--)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - one pointer may be subtracted from another of the same type—this particular operation is appropriate only for two pointers that point to elements of the same built-in array

Pointer Expressions and Pointer Arithmetic

- Assume that `int v[5]` has been declared and that its first element is at memory location 3000.
- Assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is 3000).
- Figure 8.15 diagrams this situation for a machine with four-byte integers. Variable `vPtr` can be initialized to point to `v` with either of the following statements:

```
int *vPtr = v;
int *vPtr = &v[0];
```

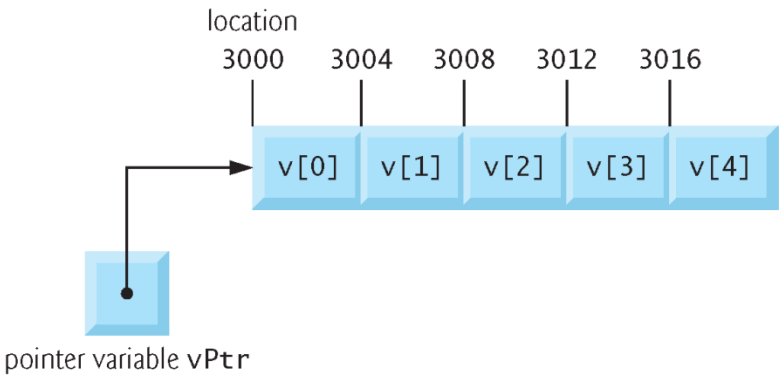


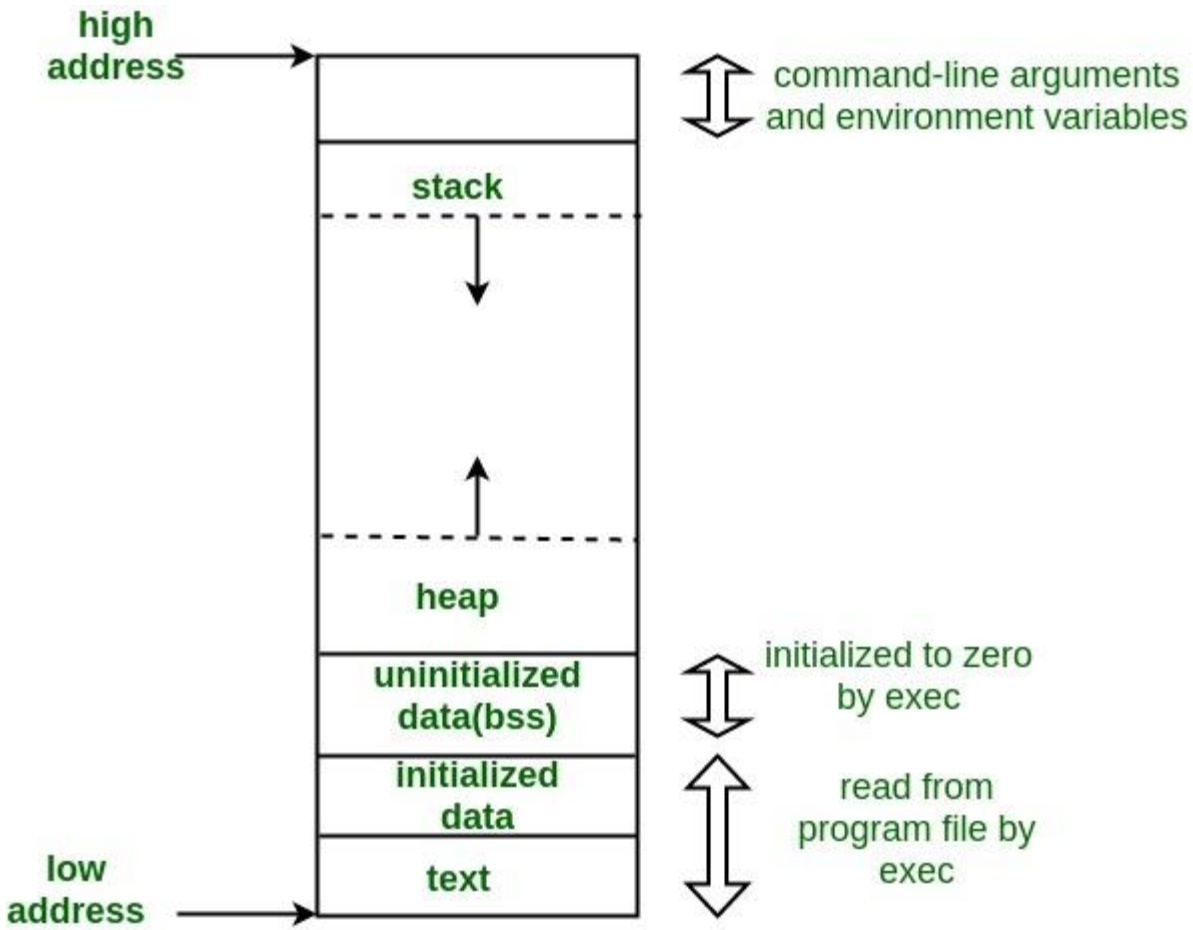
Fig. 8.15 | Built-in array `v` and a pointer variable `int *vPtr` that points to `v`.

Pointer Expressions and Pointer Arithmetic (cont.)

Adding Integers to and Subtracting Integers from Pointers

- In conventional arithmetic, the addition `3000 + 2` yields the value `3002`.
  - This is normally not the case with pointer arithmetic.
  - When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer *times the size of the object to which the pointer refers*.
  - The number of bytes depends on the object's data type.

Memory Layout of a Running C/C++ Program



Struct Declaration

```
struct Student          // no memory reserve yet
{
   int age;
   string name;
   double gpa; // grade point average
};

int main()
{
  Student jack;     // Reserve memory storage for jack
  jack.age = 18;
  jack.name = "Jack Smith";
  jack.gpa = 2.89;
  Student jill = {20, “Jill Doer”,  3.57};   // Reserve mem storage for jill
  Student  students[20];  // array of Student
}
```