Pointers

Michael R. Nowak
University of Illinois Urbana-Champaign

Lesson Outline

1. Review
2. Pointers
3. Important to note

Lesson Outline

1. **Review**
2. **2.** Pointers
3. Important to note

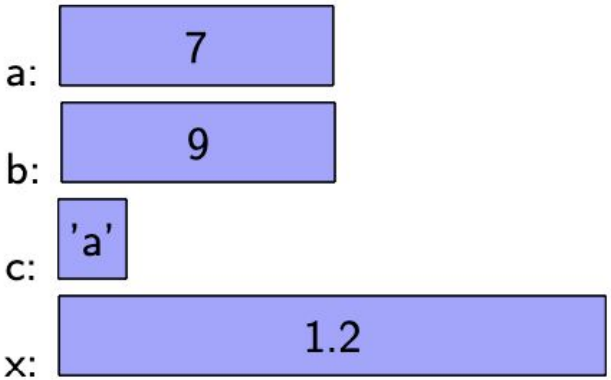
Thinking about objects, types, and values

Type	*Defines a set of possible values and a set of operations for an object*
Object	*Memory that holds a value of a given type*
Value	*Set of bits in memory interpreted according to type*
Variable	*Named object*

Thinking about objects, types, and values

A program variable is an abstraction of a computer memory cell or collection of program memory cells:

```
int a = 7;
int b = 9;
char c = 'a';
double x = 1.2;
```



Note: different types of objects take up different amounts of space

Variables

Programmers often think of variables as names for memory locations, but there is much more to a variable than just a name:

Name	*composed of a sequence of letters and digits*
Address	*is the machine memory address with which it is associated*
Type	*determines the range of values stored and set of operations*
Value	*variable is the contents of the memory cell or cells associated with the variable*
Scope	*the part of the program in which a name has a particular meaning*
Lifetime	*bound object's lifespan from the point of its allocation to deallocation*

Declaration of variables

- A declaration is comprised of four parts:
  - An optional specifier
    - An initial keyword that specifies some non-type attribute
    - E.x., `const`
  - A base type
  - A declarator
    - Composed of a name and optionally some declarator operators that are either prefix or postfix; most common declarator operators include:

*	pointer	prefix
*const	constant pointer	prefix
&	reference	prefix
[]	array	postfix
()	function	postfix

    - Postfix declarator operators bind more tightly than prefix ones
  - An optional initializer

Lesson Outline

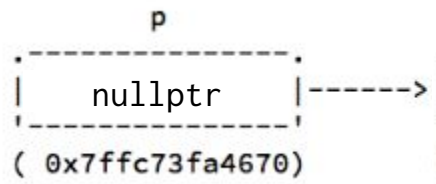
1. Review
2. **Pointers**
3. Important to note

Pointers

Let's declare our first pointer; it will point to nothing.

```
int* p = nullptr; // declares p as a pointer to an integer object
                  // and initializes p with nullptr
                  // meaning p points to nothing
```

We can visualize this informally as:



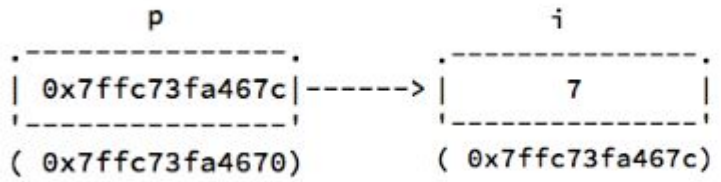
Pointers

Pointers are objects in their own right, so we can assign them values.

```
int* p = nullptr; // declares p as a pointer to an integer object
                  // and initializes p with the address of nullptr
                  // meaning p points to nothing

int i = 7; // assume i's address 0x7ffc73fa467c
p = &i;    // assigns the address of i to p; p now points to i.
```

We can visualize this informally as:

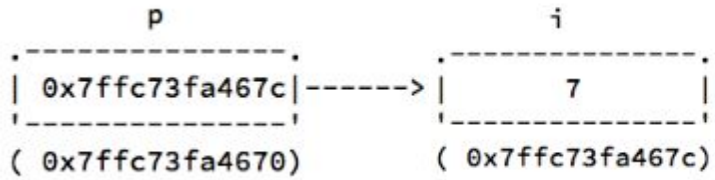


Pointers

Let's now declare our first pointer that actually "starts out" pointing to *something*

```
int i  = 7;  // assume i's address 0x7ffc73fa467c
int* p = &i; // declares p as a pointer to an integer object
             // and initializes p with the address of i
            // therefore, p points to i
```

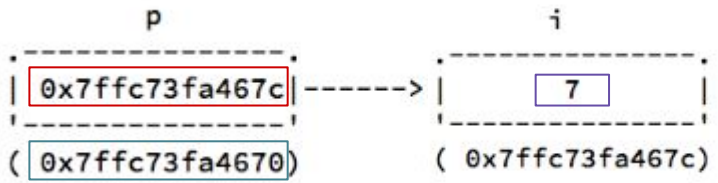
We can visualize this relationship informally as:



Pointers

Given

```
int i  = 7;
int* p = &i;
```




```
std::cout << &p << ' ' << p << ' ' << *p << std::endl;
```

Outputs the address of the pointer, the pointer's value, and the value stored in the object being pointed to:

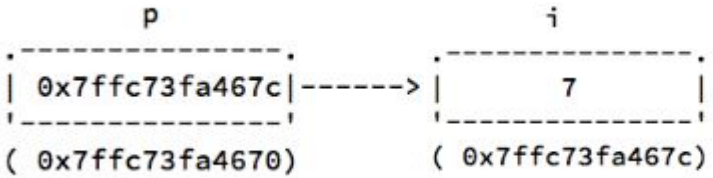
```
0x7ffc73fa4670 0x7ffc73fa467c 7
```

Pointers: assignment to the object pointed to

Given that

```
int i = 7;
int* p = &i;
```

Establishes the following relationship



For the following statement

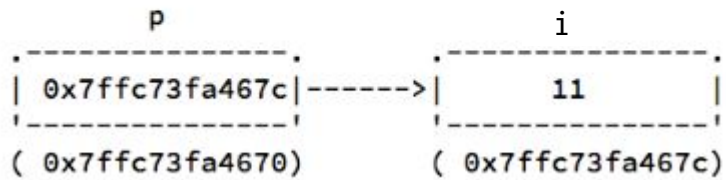
```
*p = 5 + 6;
```

Initialization/assignment is right-to-left associative, so `5 + 6` is evaluated to `11` first.

`*p =` is evaluated next:

1. go to the address `0x7ffc73fa467c`
2. isolate the integer object that starts at that address
3. and assign the integer value `11` into that object

Now,

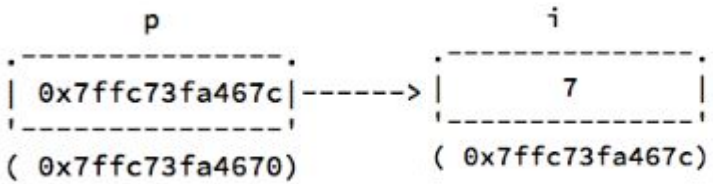


Pointers: using the value stored in object pointed to

Given that

```
int i = 7;
int* p = &i;
```

Establishes the following relationship:



For the following statement,

```
int j = *p;
```

Initialization/assignment is right-to-left associative, so `*p` is evaluated first:

1. go to address `0x7ffc73fa467c`
2. begin interpreting the data there as an `int`eger
3. this evaluates to 7

The statement now reads,

```
int j = 7; // b/c *p evaluates to 7
```

So, `j` starts out with the value of `7`.

Lesson Outline

1. Review
2. Pointers
3. **Important to note**

Important to note:

& and * are used both as an operator in expressions and as part of the declarator to form compound types

Make sure that you understand that it is the context in which these symbols are used that determines their meaning

```
int i  = 11;
int& r = i;
```

Here, & appears as a declarator operator: a reference is being created

```
int i  = 11;
int* p = &i;
```

Here, * appears as a declarator operator: a pointer is being created; & appears in the initializer expression and is the *address-of operator*

```
int i  = 11;
int* p = &i;
*p = 2;
```

Here, * appears as a declarator operator and the dereference operator (give me the object being pointed to).