Dereferencing Pointers

Let's look at the list of pointers on the previous page again.

```
int x{42}, y{0}, a[10]; // x->int, y->int, a->array

int *p1{&y};          // points to y
int *p2{&x};          // points to x
int *p3{new int{3}}; // points to int on heap
int *p4{a};           // points to first element of a
int *p5{a+10};        // points "one past" the array a
int *p6{nullptr};     // points to "nothing"
int *p7;              // uninitialzed (invalid)
```

The `*` **dereferencing operator** returns the value that a pointer points to, **provided that** the pointer points to a **valid object**, such as `p1` and `p2`. Using the dereferencing operator on `p5`, `p6` or `p7` produces **undefined behavior**. The value that a pointer "points to" is called its **indirect value**.

Since `p1` is a pointer to `int`, the compiler "knows" that `*p1` **must be an integer object**. Thus `*p1` turns out to be **another name (or alias) for the variable** `y`. Like the simple name `y`, `*p1` is an ***lvalue*** , and you can assign new values to it.

```
int x{42}, y{0};
int *p1{&y};         // points to y
int *p2{&x};         // points to x
*p1 = 17;            // assign to indirect value
```

This last statement changes the value in the variable `y` because that is the target of the pointer `p1`. `p1` **is unaffected** by this assignment; it continues to point to the variable `y`. Click the little running-man on the left to see this animated in a new window.