Now you C the point!: Making sense of pointers in C

Background: Why are C pointers hard to understand?

To understand a C program, one needs to be equipped with a clear mental model of how memory is represented and manipulated by the program. The model need not be exactly how memory is represented in hardware. Instead, it suffices to build a symbolic or visual, but robust model of memory.

Without a clear mental model, understanding pointers can be tricky. The beginning student of C is also burdened with at least least two other hurdles: C's awkward syntax and its confusing semantics. The field of semantics is concerned with building *mental models* of how a program runs, and, in the case of C, how it represents and manipulates memory. So far, it has been considered hard to build and visualise clear mental models of C programs and there is no standard way of doing so. This is particularly acutely felt when declaring and using pointers.

The unintuitive syntax of C

Consider the declaration `int x`. We would like to think of `int` as the set of all integers and `x` as a variable denoting a member of that set, in which case `x int` would make more sense. It gets worse when we have pointers. How do we interpret `int *p`? One way is to consider navigating (again, right to left!) this declaration by starting from `p`, traversing the `*` and ending up at `int`. This idea of navigation is central to the model we present, but unless carefully defined, it doesn't always work!

C violates basic reasoning with equality

Here is an example that shows how simple equational reasoning fails in C. Consider the C fragment

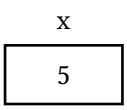
```
int x = 5;
int y = 5;
```

Clearly, `x` and `y` both now denote the value `5` according to the semantics of C. We write this as $x \underset{C}{=} y$. Now think of `&` as an operator, that takes a variable and returns its address. So `&x` returns the address of `x`. So we have $x \underset{C}{=} y$ but $\&x \underset{C}{\neq} \&y$. This is counter-intuitive because it violates a fundamental principle of mathematical reasoning:

Principle of Substitution: If $e_1 = e_2$, then, in any expression e containing e_1, replacing e_1 with e_2 in e should make no difference.

Traditional Box and pointer models of C

The traditional mental model for C is called the *box and pointer* model. In this model, boxes are memory locations and the boxes contain values. However, this model has its own problems, as illustrated by the following example. The C statement `int x = 5;` is represented as the box diagram



Notice that the box (an address) is itself labeled `x`. This results in conflating `x` with its address. There is no way to distinguish `x` from `&x`, namely the address of `x`. So, `printf("%p", &x);` will print a value that is neither `x` nor `5`.

Graph Model

We propose a new mental model for understanding pointers in C. This model is motivated by the need to be able to do simple mathematical reasoning involving function application and mathematical equality. Our mental models are now represented as *graphs*.

Graphs and Functions

A graph is simply a collection of vertices and arrows between the vertices. In addition, each vertex and arrow is labelled. Graphs are used to represent a variety of things in computer science. We use a graph to represent function application. So, if $f(x) = y$, then this is represented as an arrow from x to y labelled f.

A path corresponds to a composition of function applications.

The state of the memory is modelled as a graph. This graph evolves as each statement is executed. The model may be described by the following set of rules (To keep our model simple, let us assume we only have `int` as the primitive type)

1. **Vertices:** Vertices have labels. There could be multiple vertices with the same label.
2. **Vertex labels:** A vertex label is one of the following:
   - *Variable*: x, y, z, p, etc.
   - *Value*: one of
   - *Variable address*: a_x, a_y, etc.
   - *Memory address*: m_0, m_1, etc.
   - *Integer*: $0, 1, -1$, etc.
   - *Undefined*: $\perp$ (also called `bottom`)
3. **Vertex classification:** Vertices are classified into
   - **Variable vertex:** a vertex whose label is a variable.
   - **Address vertex:** a vertex which is the source of a * arrow. It is labelled either by a variable or a memory address.
   - **Value vertex:** a vertex which is the destination of a * arrow. It is by a value label (See 6.)
3. **Variable vertices represent program variables**: For every program variable x, there is a unique variable vertex, whose label is x.
5. **& Forward Arrow and Variable Address vertices:** For every variable vertex labelled x, there is an edge labelled & emanating from that vertex to a unique address vertex labelled with the address a_x. Such a vertex, which is the destination of a & forward arrow is called a *variable address vertex*.
6. **Value label:** A value label is either an integer, a variable address, a memory address, or undefined ($\perp$).
7. *** Arrow and Address Vertices:** There is an arrow labelled * from every address vertex to a value vertex. This models the intuition that the address *stores* a value.
8. **& Back Arrow:** for every arrow labelled * there is a *back* arrow labelled & from the value vertex to the corresponding address vertex. This captures the notion that the value is contained or pointed to by the address. **To reduce clutter, this arrow is not displayed.**
9. **=> Arrow:** For each value vertex labelled with an address, there is an edge labelled => to an address vertex with the same label.
10. **`r` Arrow:** For every variable there is an arrow from the variable labelled `r` to a value vertex, obtained as the composition of the arrows & and *. This denotes the notion that a variable has a value.

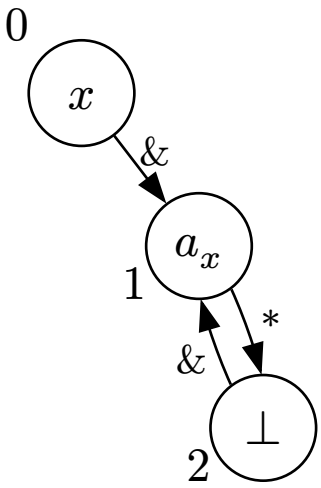
The memory graph evolves as the C program statements execute. The best way to understand the path model is through examples.

Example 1: A simple declaration

Consider the program fragment

```
int x;
```

The initial graph has three nodes: the variable x, its address a_x and the (initial) value `undefined`.

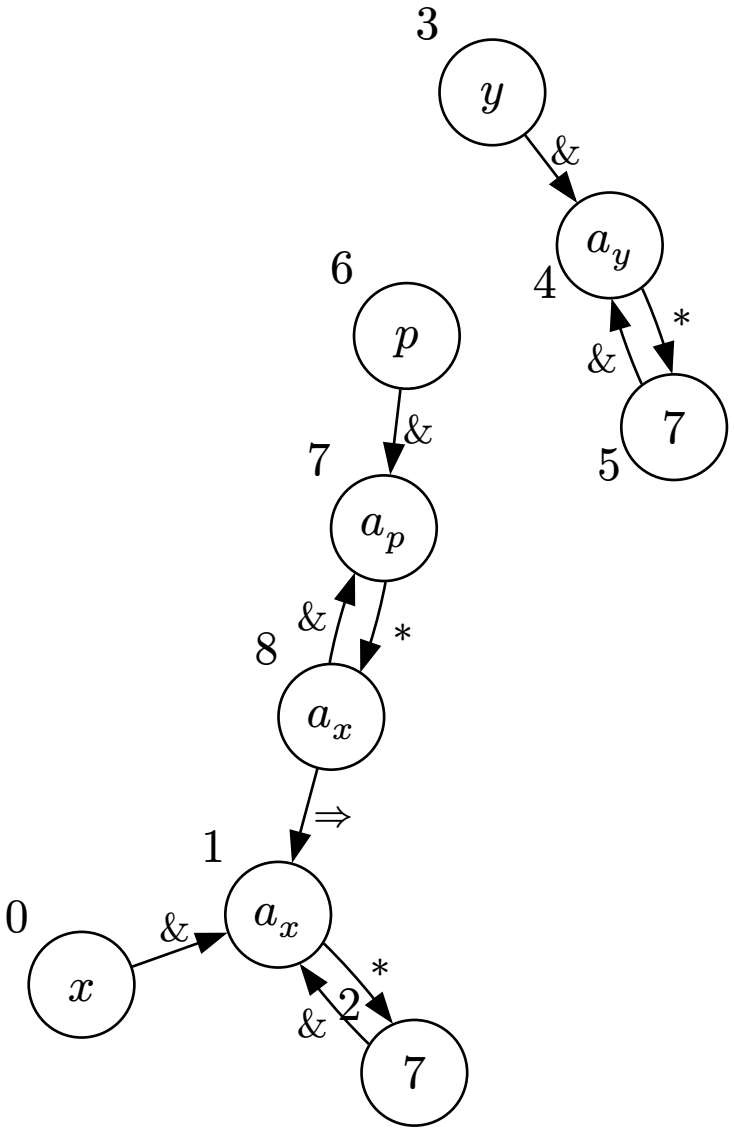


Note the single variable address vertex is denoted in blue.

Example 2: Pointer declaration, address-of and assignment

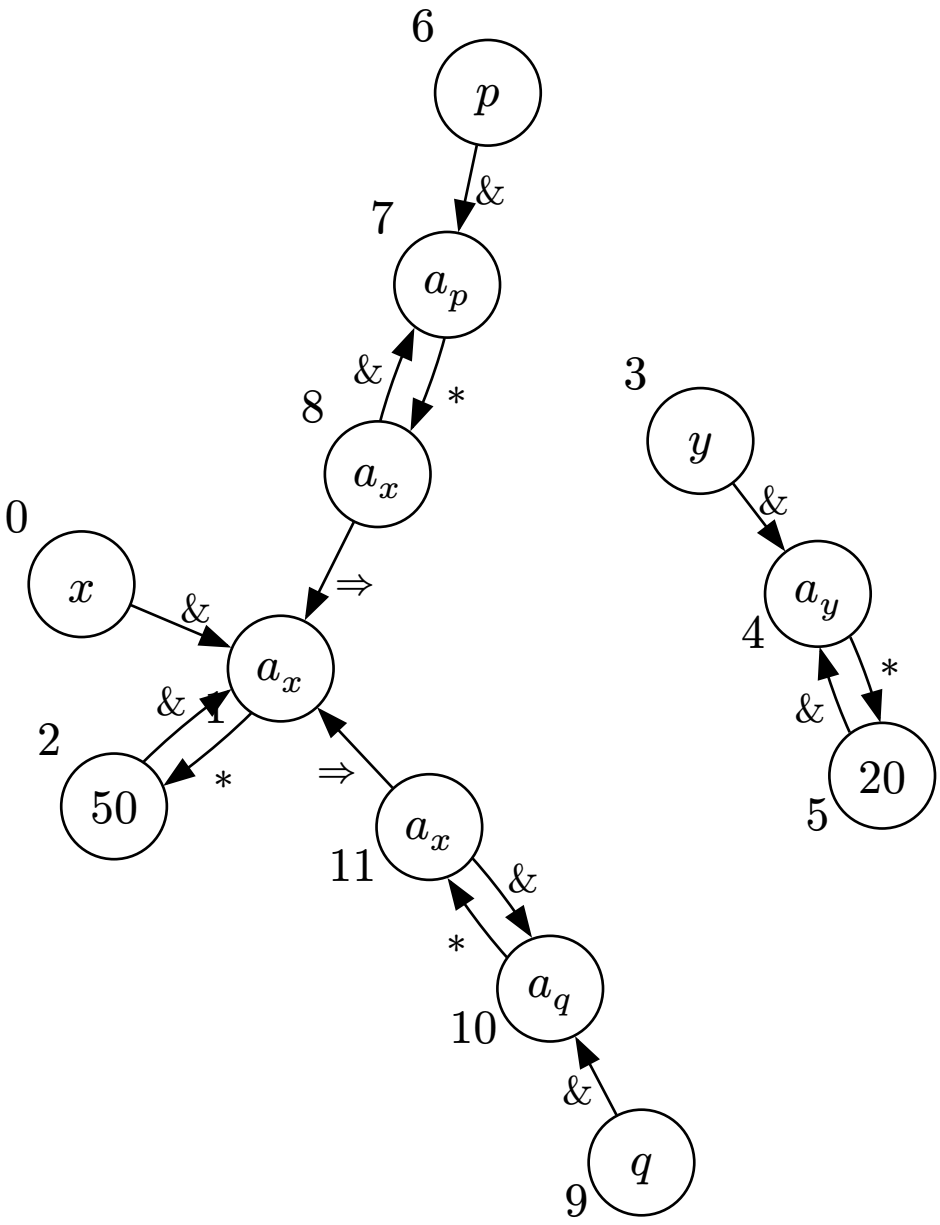
```
int x = 5, y = 7;
int *p = &x;
*p = y;
```

As the initial step, small triangular graphs created for the declarations of each program variable: `x`, `y` and `p`.



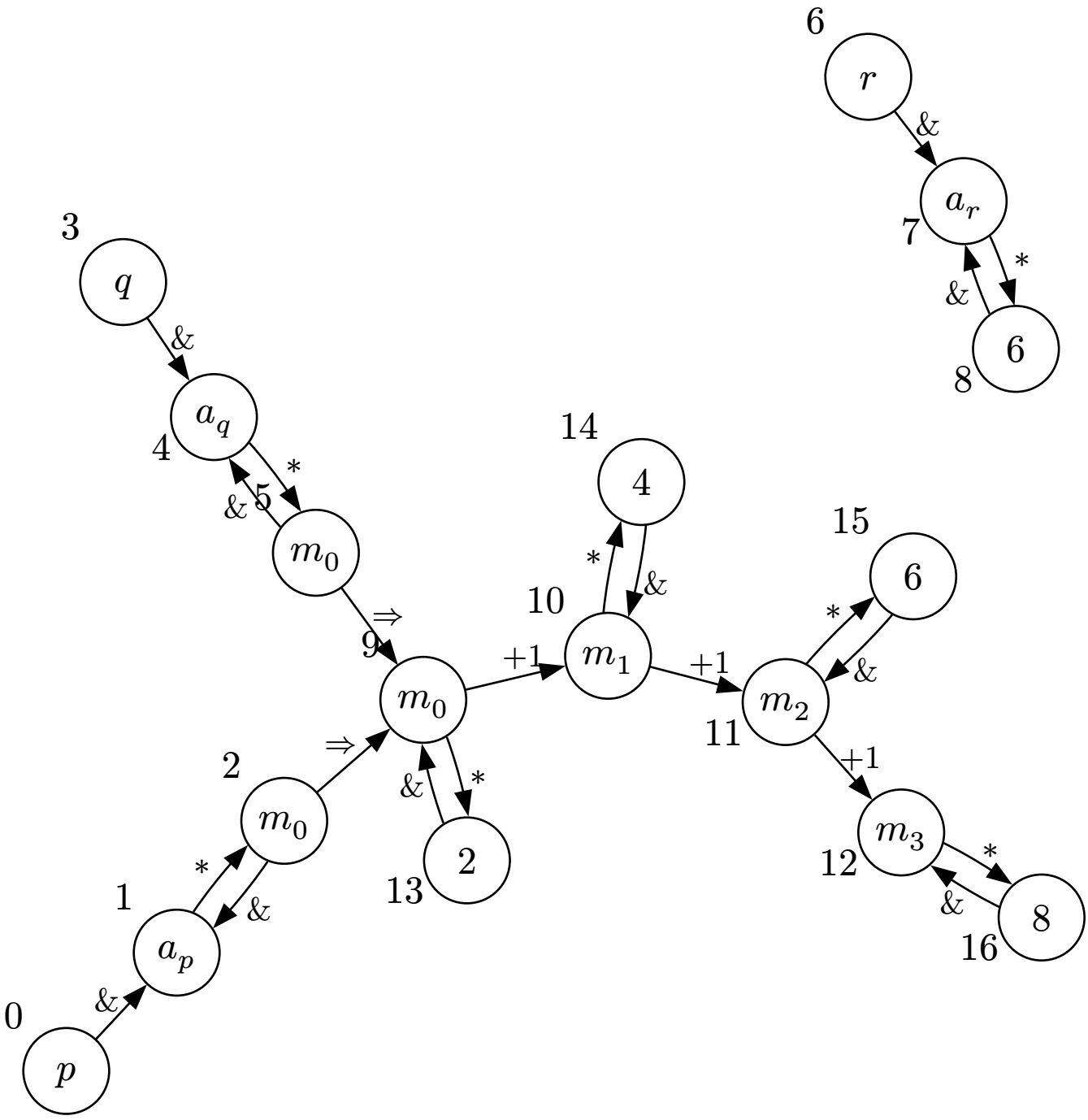
Example 3

```
int x = 10, y = 20;
int* p = &x;
int* q = &y;
*p = *q;
q = p;
*q = 50;
```



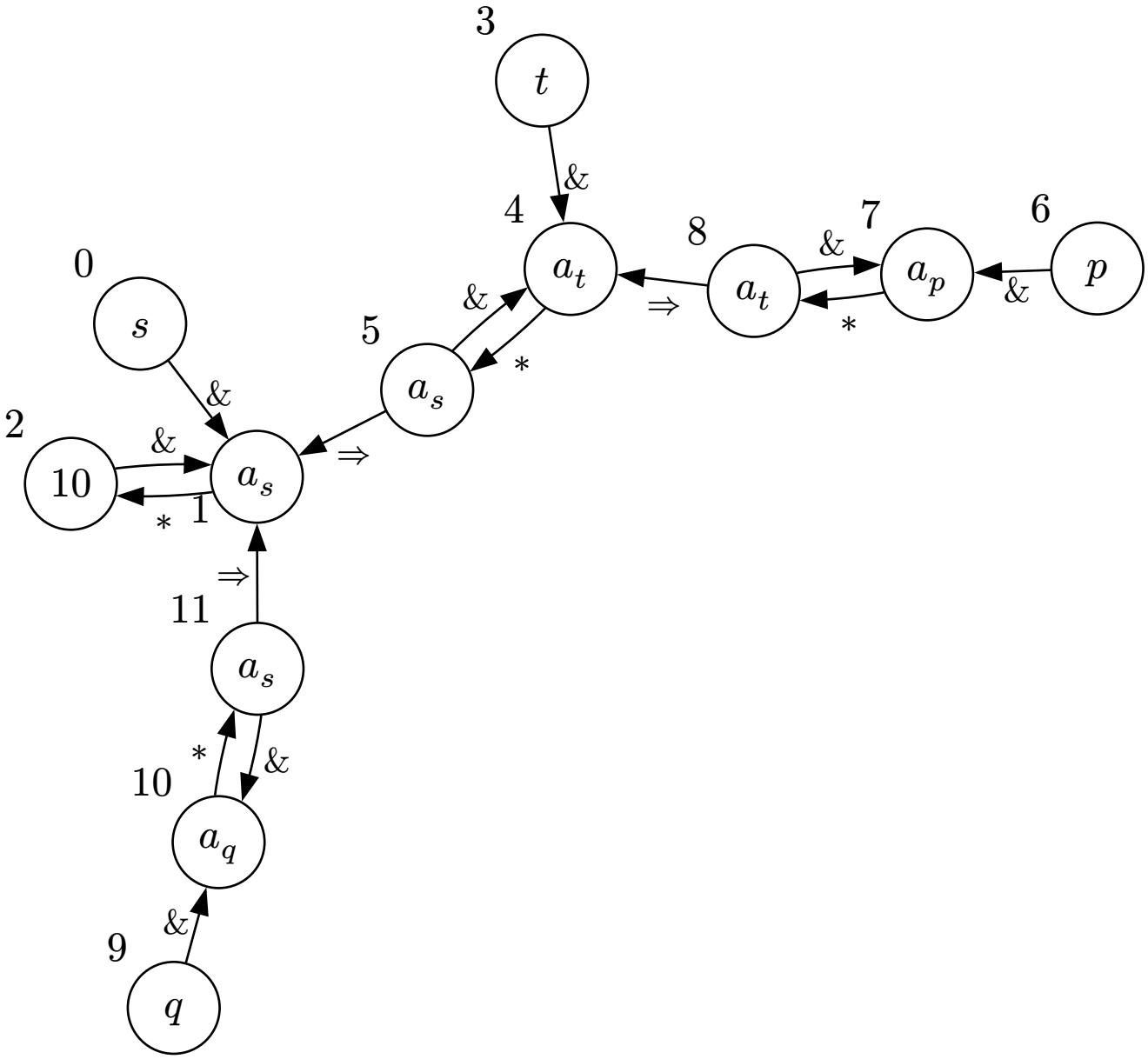
Example 4

```
int p[] = {2, 4, 6, 8};
int *q = p;
int r = *(q + 2);
```



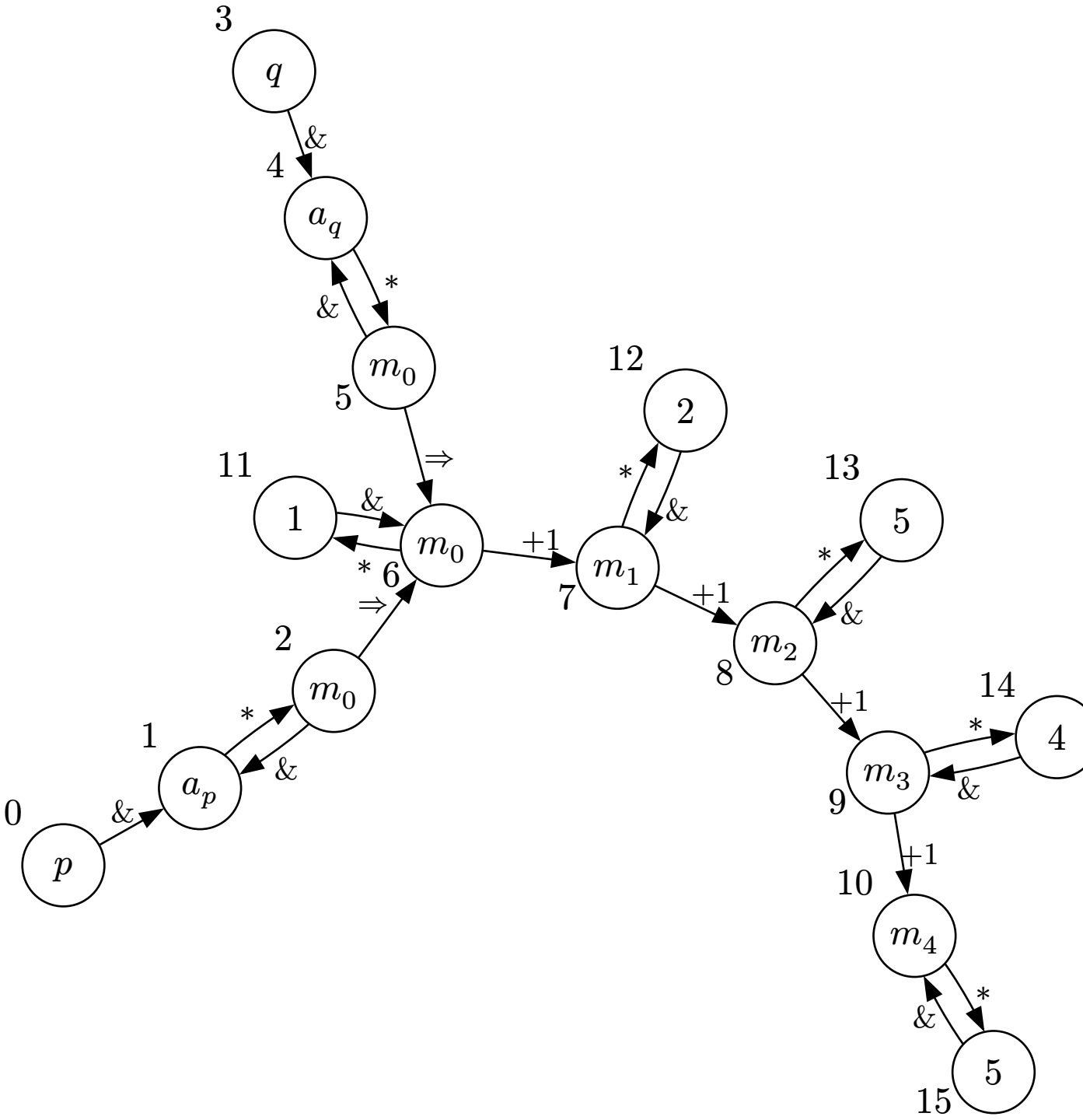
Example 5

```
int s = 3;
int *t = &s;
int **p = &t;
int *q = *p;
*q = 10;
```



Example 6

```
int q[] = {1, 2, 3, 4, 5};
int *p = q;
*(p + 2) = *(p + 4);
```



Example 7

```
int p[] = {1, 2, 3};
int x = *(p + 2);
```