C the Point: Making sense of pointers in C

Background: Why are C pointers hard to understand?

Many of us have struggled to understand pointers in C. This can be attributed to at least two different reasons. First, the syntax of C, specially when declaring pointers is awkward and unintuitive. Second, most of us lack a robust mental model of C's semantics that represents our understanding of how C manipulates pointers.

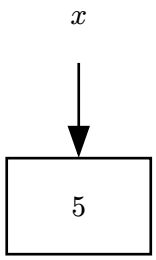
The unintuitive syntax of C

The first unintuitive thing in C, is that the type of a variable is written *before* the name of the variable. So, C insists that we write `int x` instead of the more intuitive `x int`. It gets worse when we have pointers. So, `int *p` looks strange when compared to `p *int`. One way to think of the type declaration for `p` is to consider navigating from `p` on a `*` and ending up at `int`. This idea of navigation is central to the model we present but C's syntax doesn't quite align with this model of thinking.

Traditional Box and pointer models of C

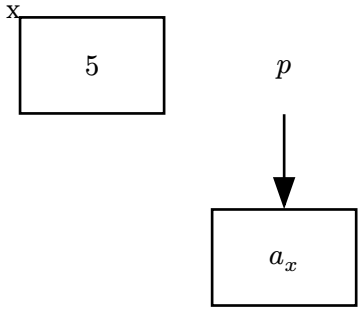
Before we introduce the new model, let us consider one of the most common mental models employed by students of C programming. It is called the *box and pointer* model.

In this model, boxes are memory locations and the boxes contain values. Unfortunately, the box and pointer diagram fails to capture adequate information to yield an unambiguous answer. Here's an example, the C statement `int x = 5` is represented as the box diagram



This states that the meaning of `x` is the box. Notice that the box itself is labeled `x`. This results in conflating `x` with its address, so, there is no way to denote `&x`, which is another value, namely the address of `x`.

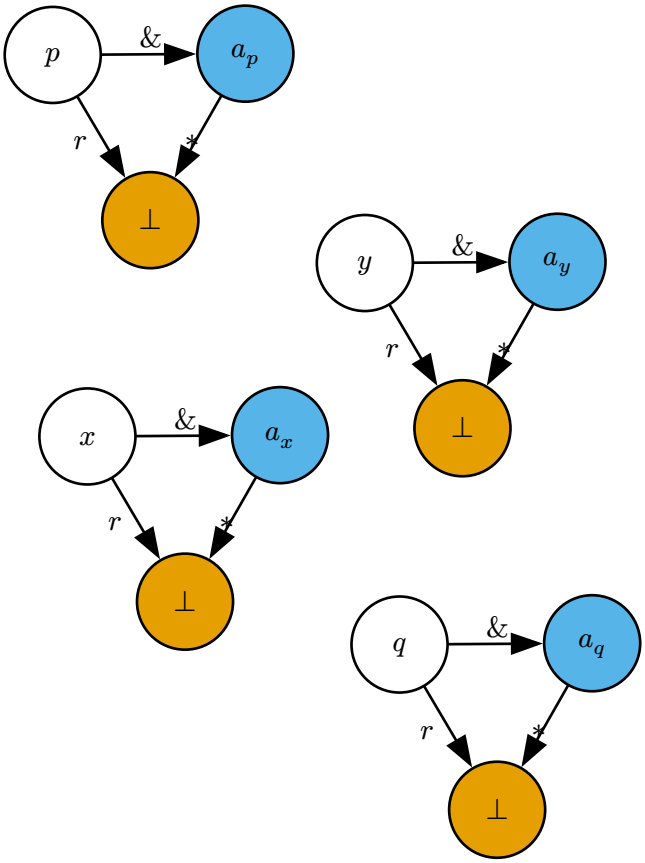
This notation conflates the variable `x` with the box (or address) it denotes because the addre Adding the statement `int *p = &x;` results in the following diagram: Now, to find the value of `*p`, we follow the pointer in the box labeled `p`. This takes us to the box containing 5. The value of `*p` is therefore 5. This looks fine until we add



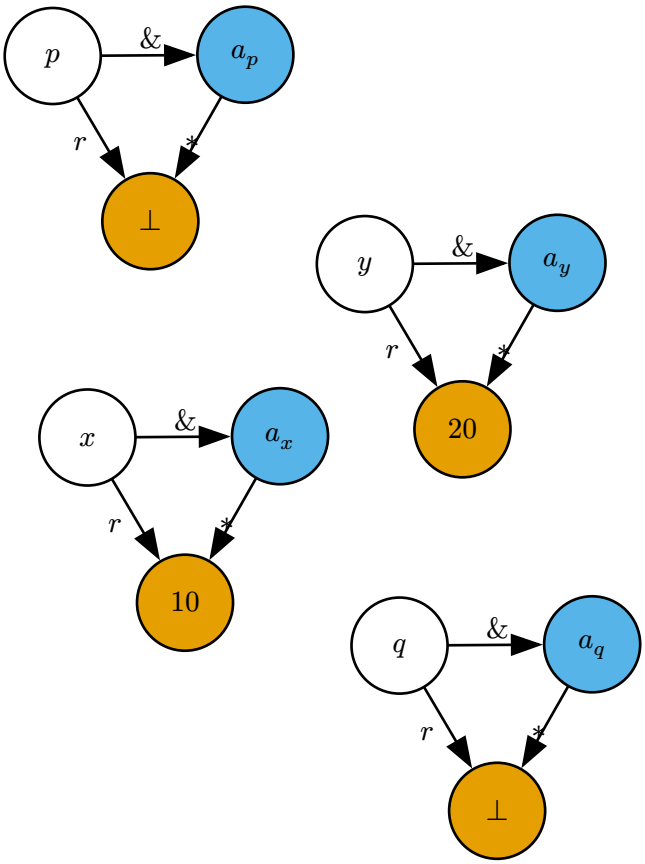
Now, imagine we wish to derive the value of `*p`, which is 5. For this, we would start from `p`, then go to the box it points to, pick up the value in the box, namely a_x, and then go to x

Example 1

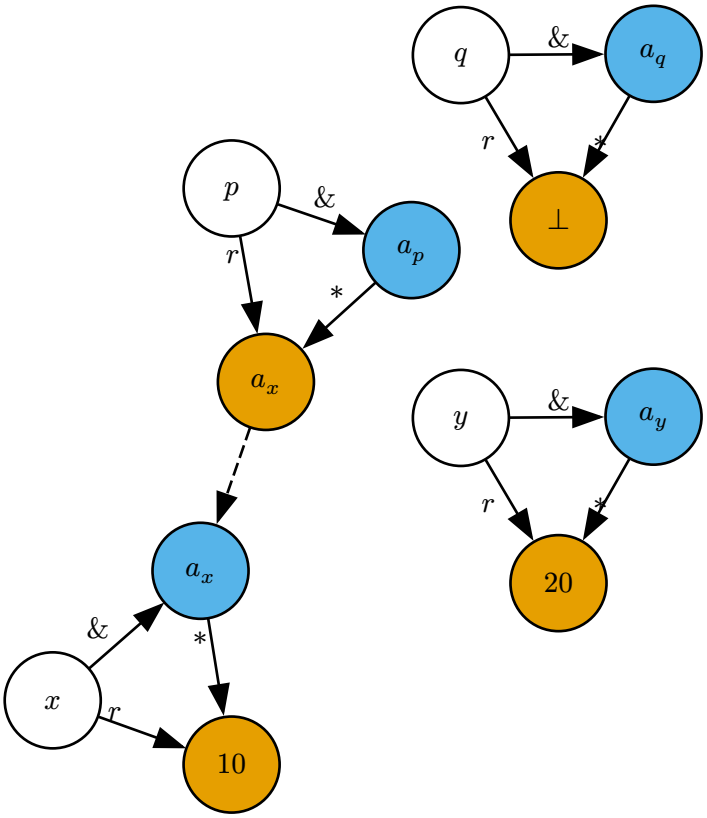
```
int x = 10, y = 20;
int* p = &x;
int* q = &y;
*p = *q;
q = p;
*q = 50;
```



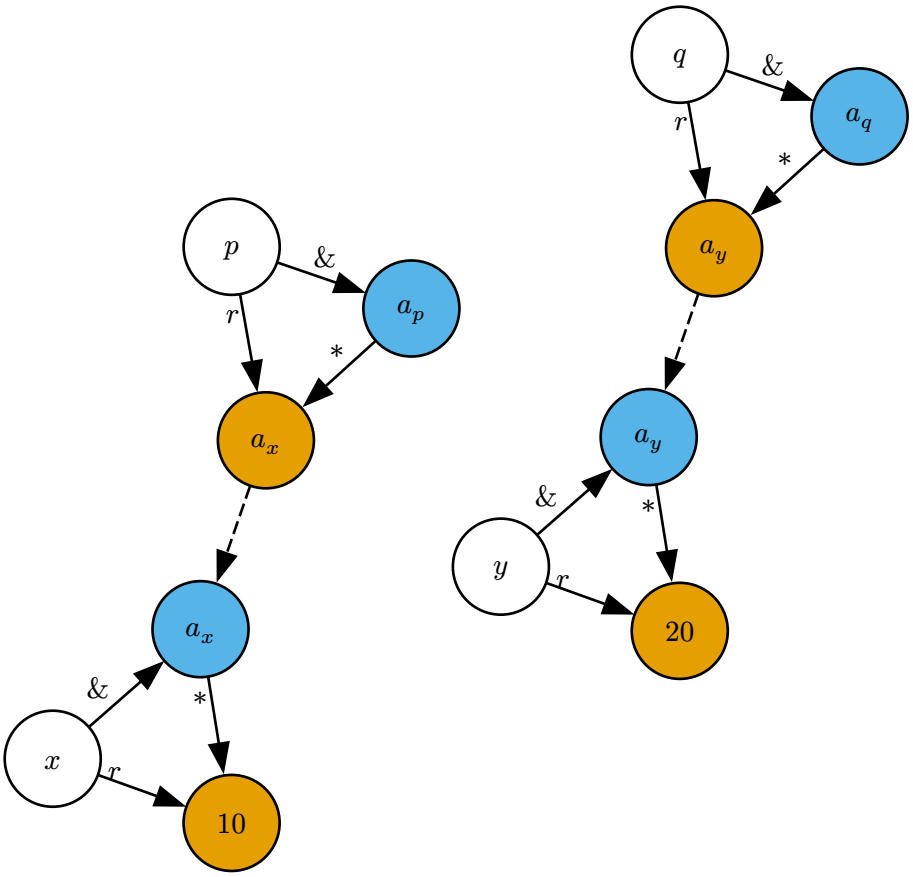
↓ int x = 10, y = 20;



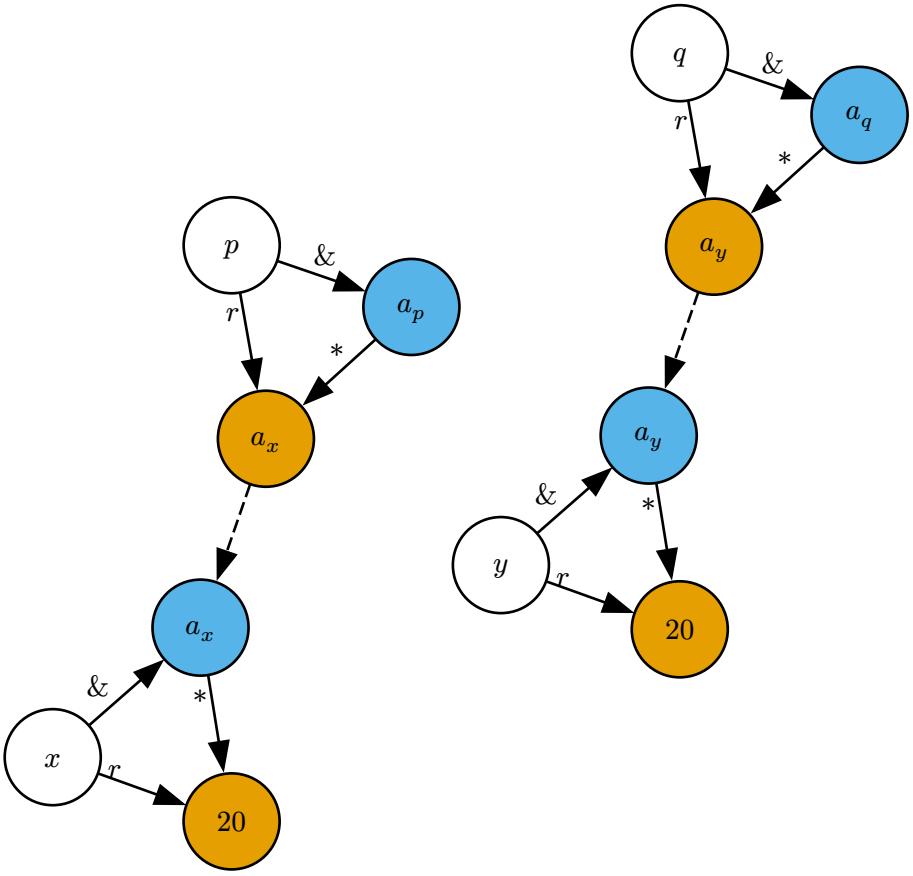
↓ int* p = &x;



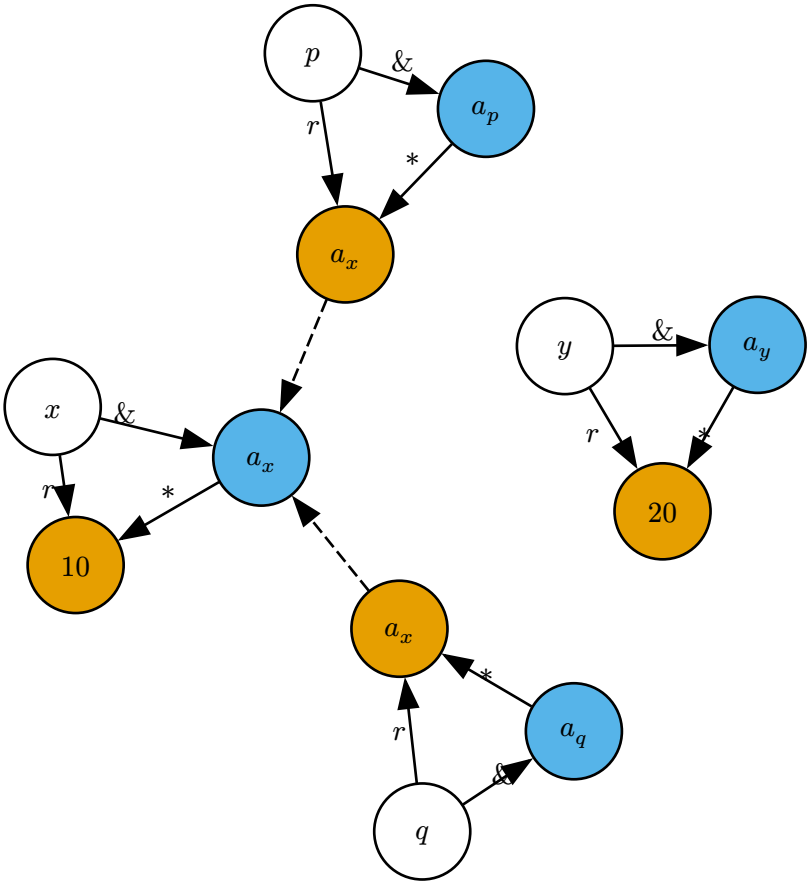
↓ int* q = &y;



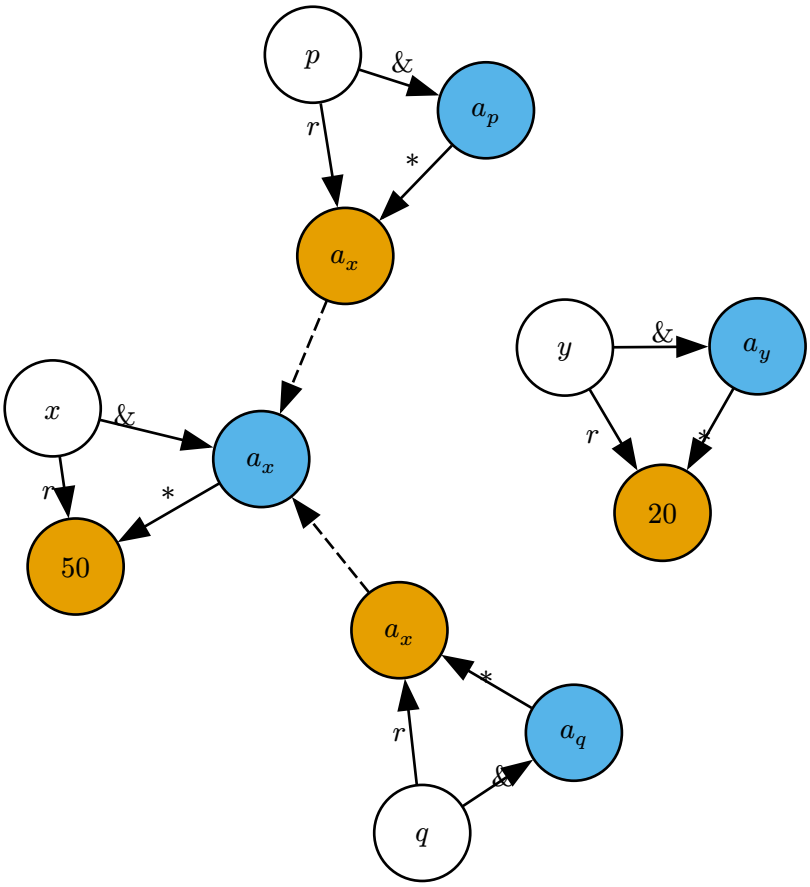
↓ *p = *q;



↓ q = p;

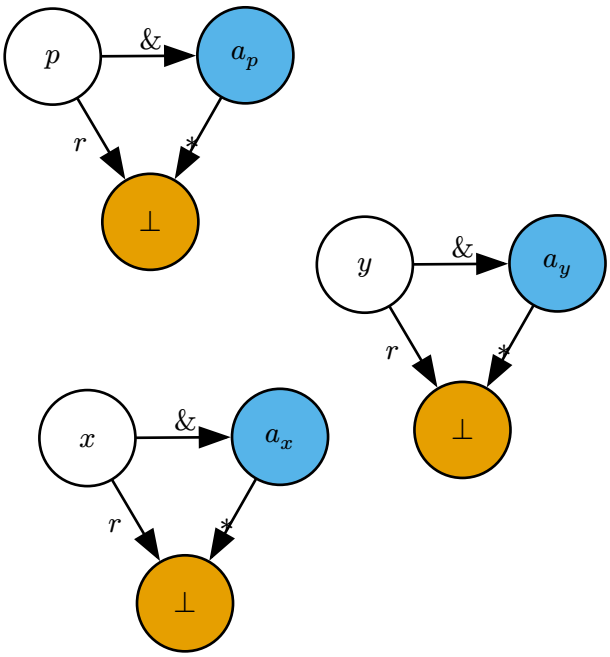


↓ *q = 50;

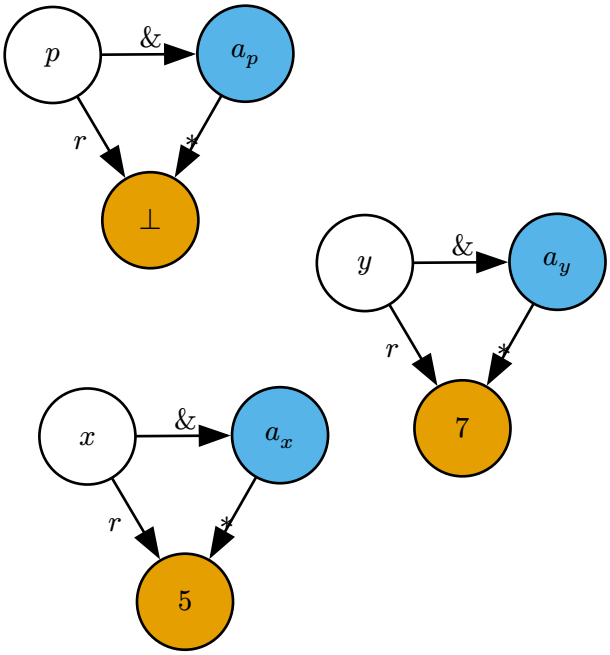


Example 2

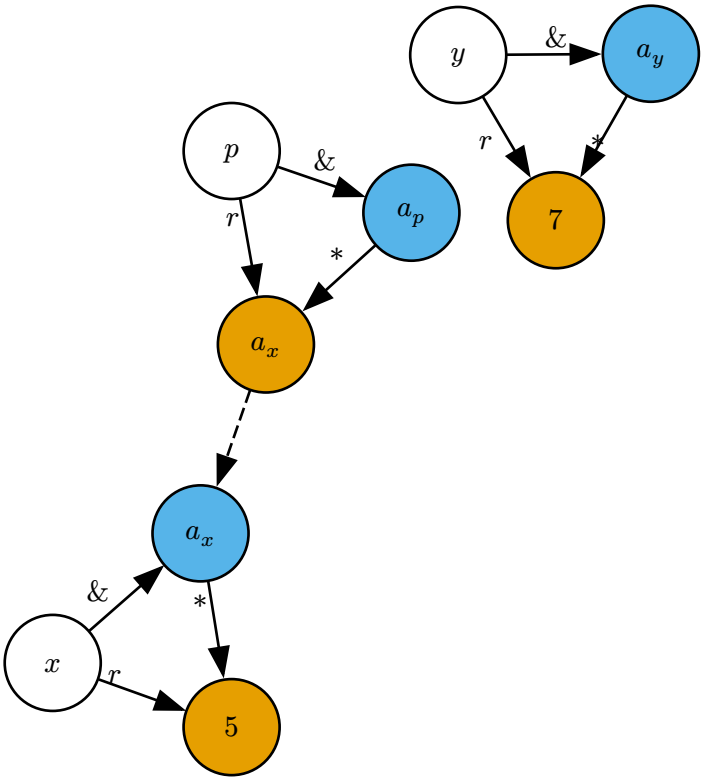
```
int x = 5, y = 7;
int *p = &x;
*p = y;
```



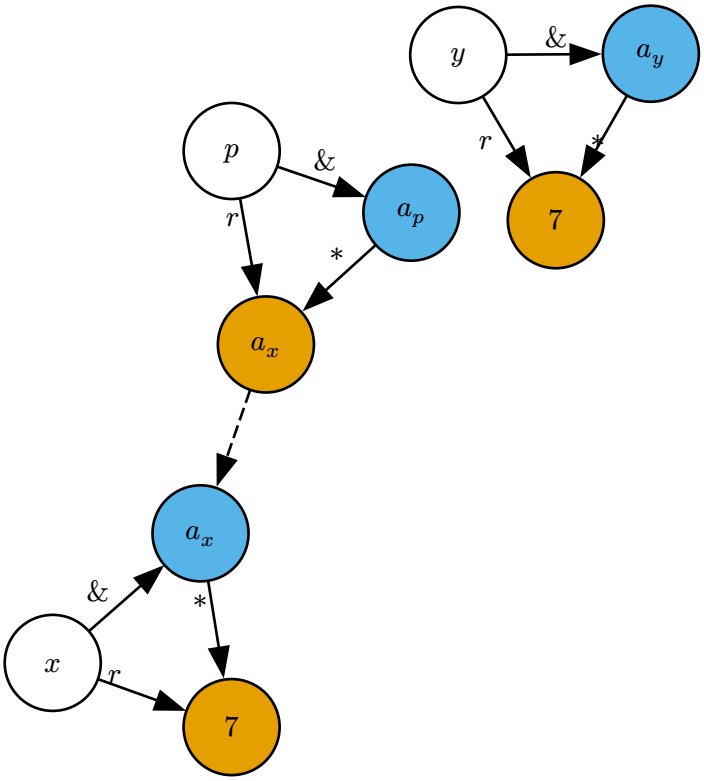
↓ int x = 5, y = 7;



↓ int *p = &x;

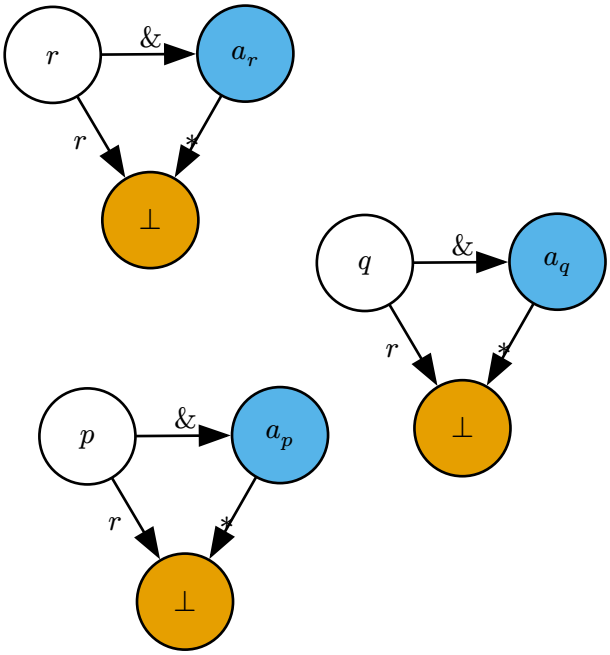


↓ *p = y;

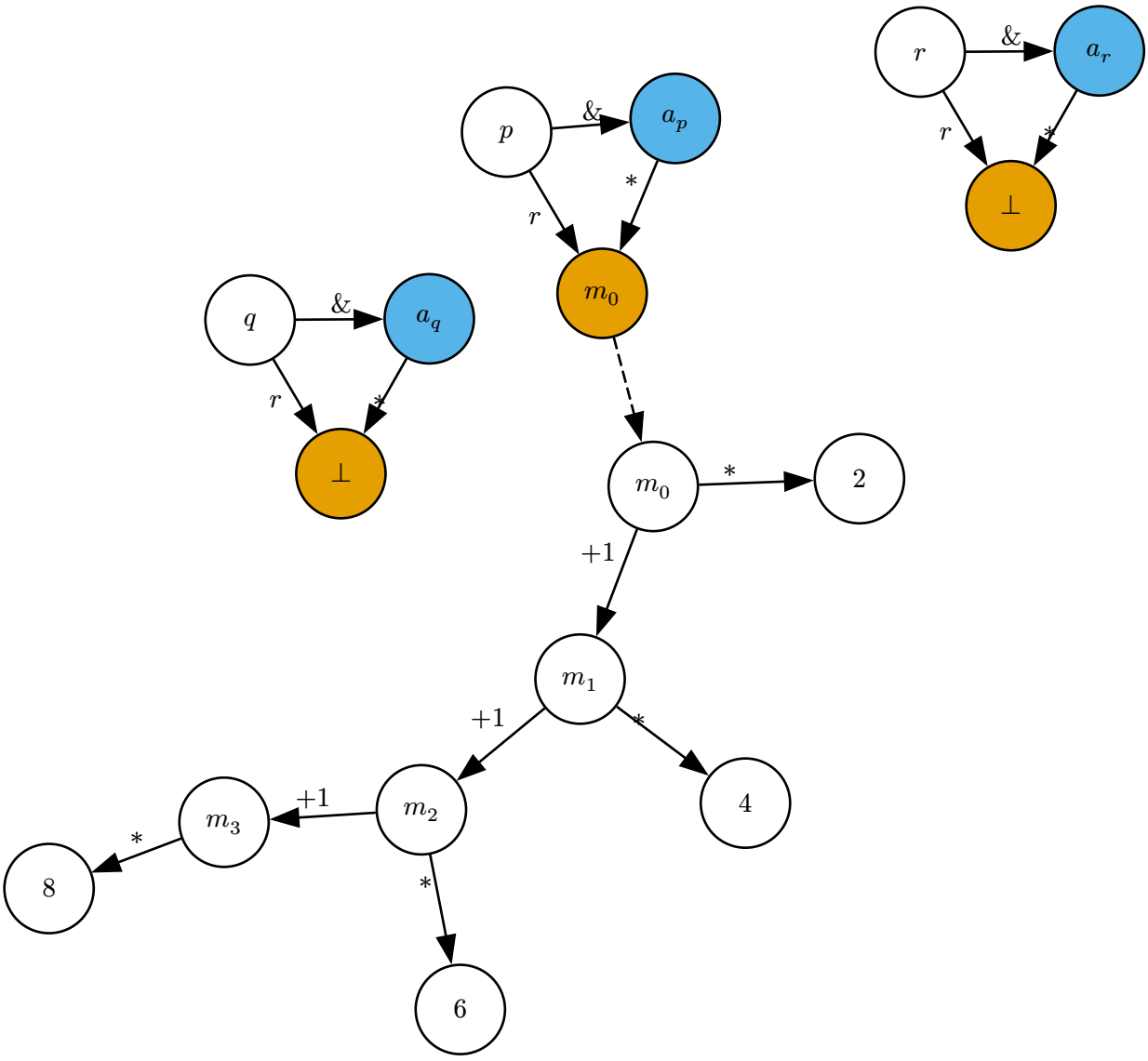


Example 3

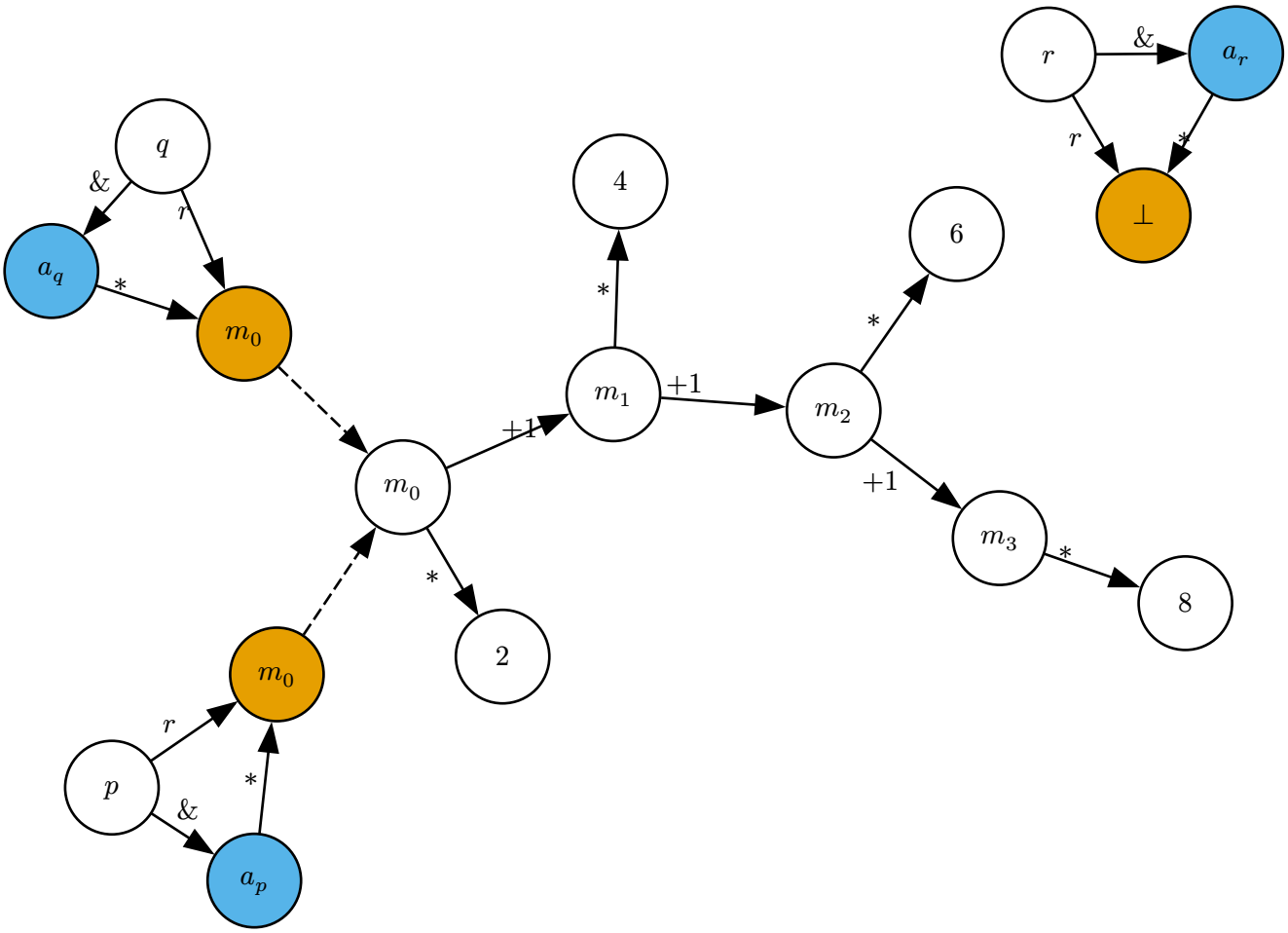
```
int p[] = {2, 4, 6, 8};
int *q = p;
int r = *(q + 2);
```



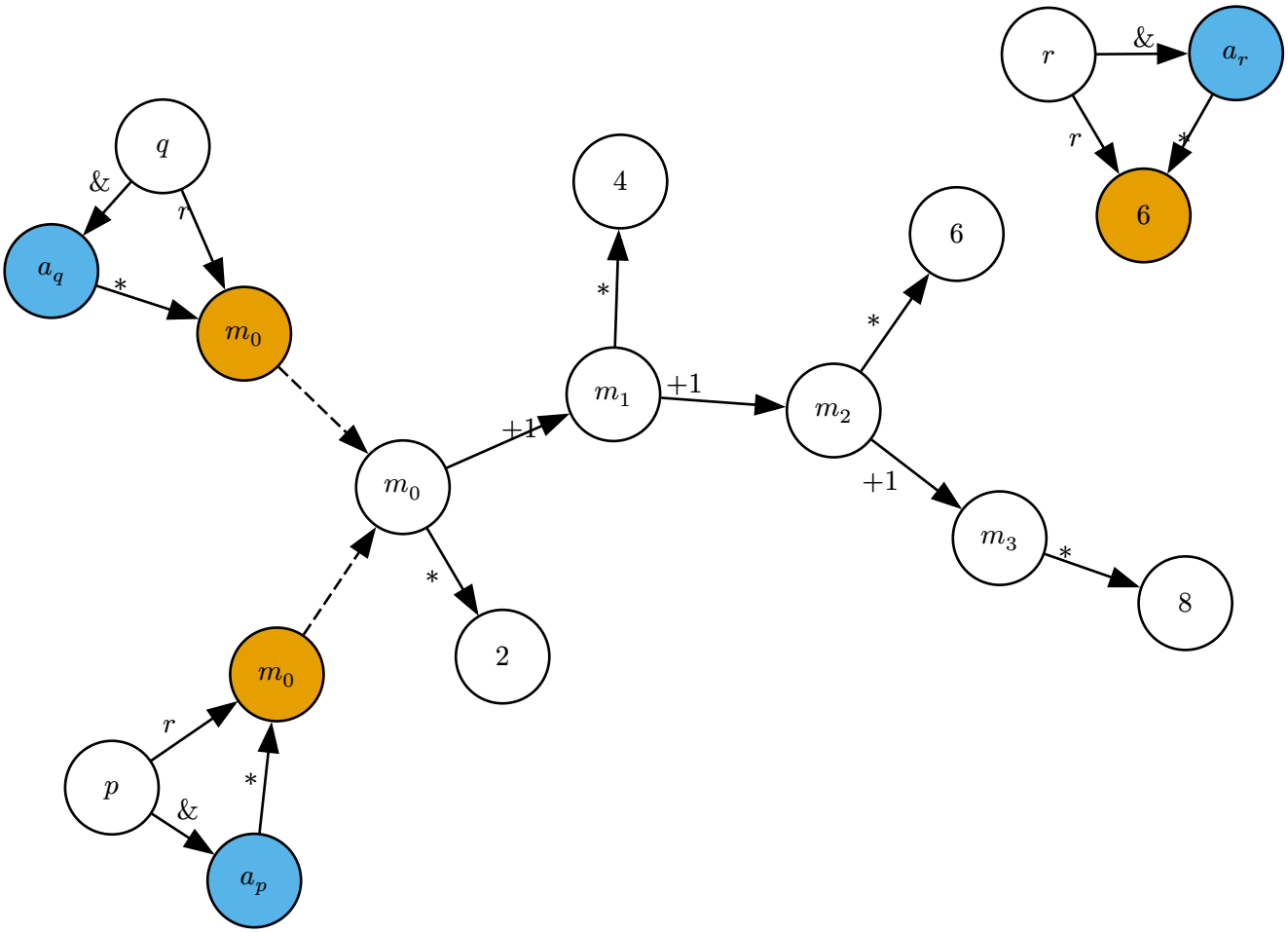
↓ int p[] = {2, 4, 6, 8};



↓ int *q = p;

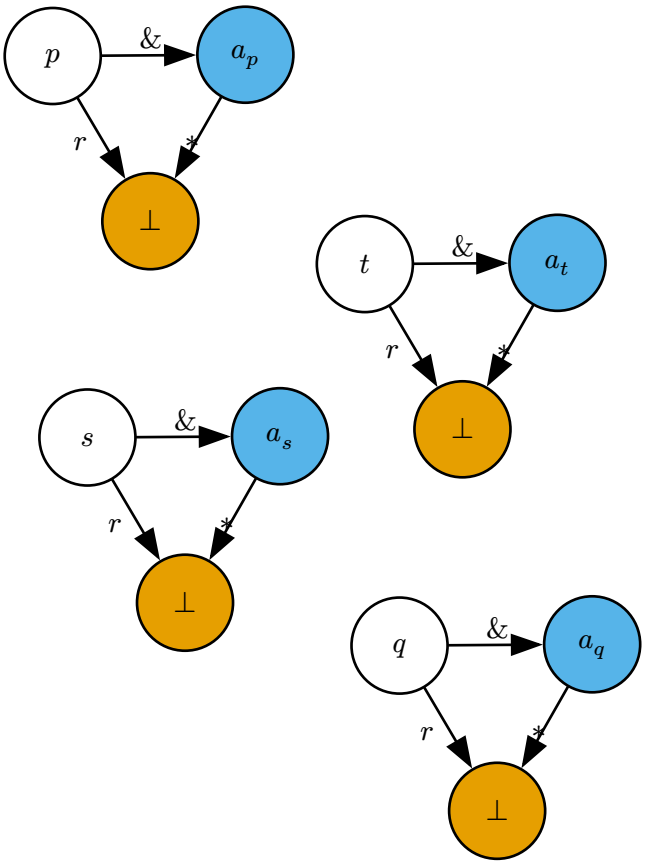


↓ int r = *(q + 2);

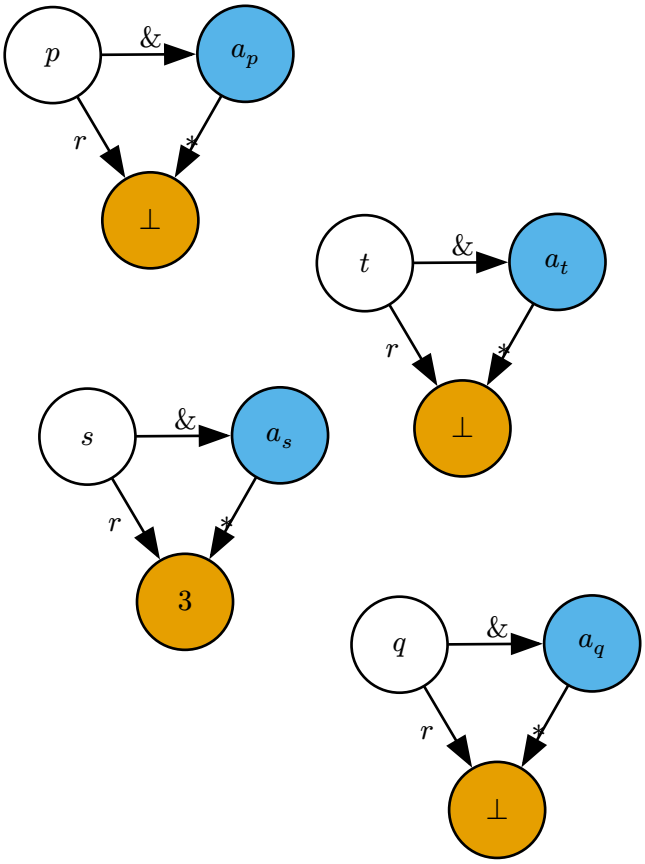


Example 4

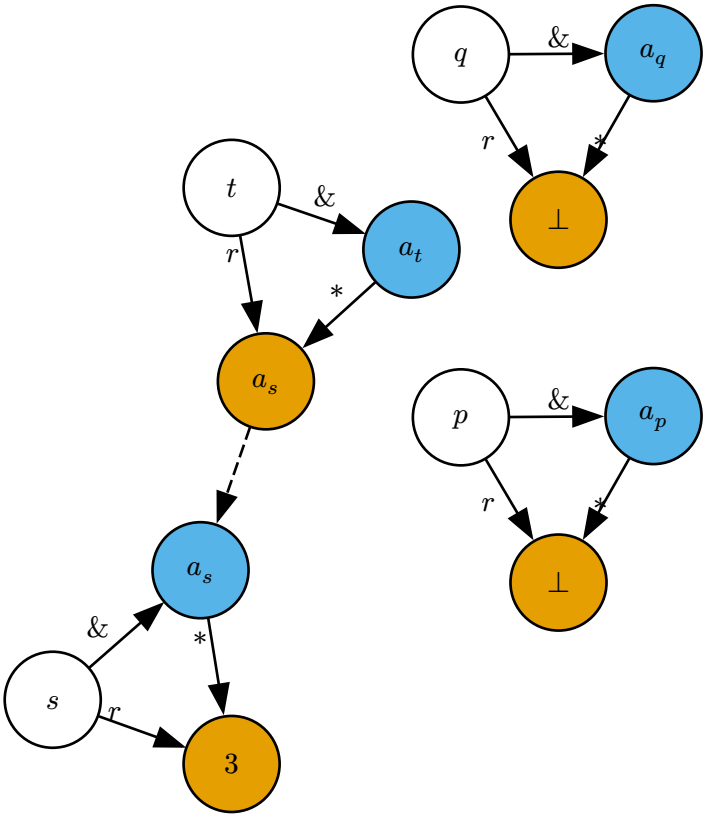
```
int s = 3;
int *t = &s;
int **p = &t;
int *q = *p;
*q = 10;
```



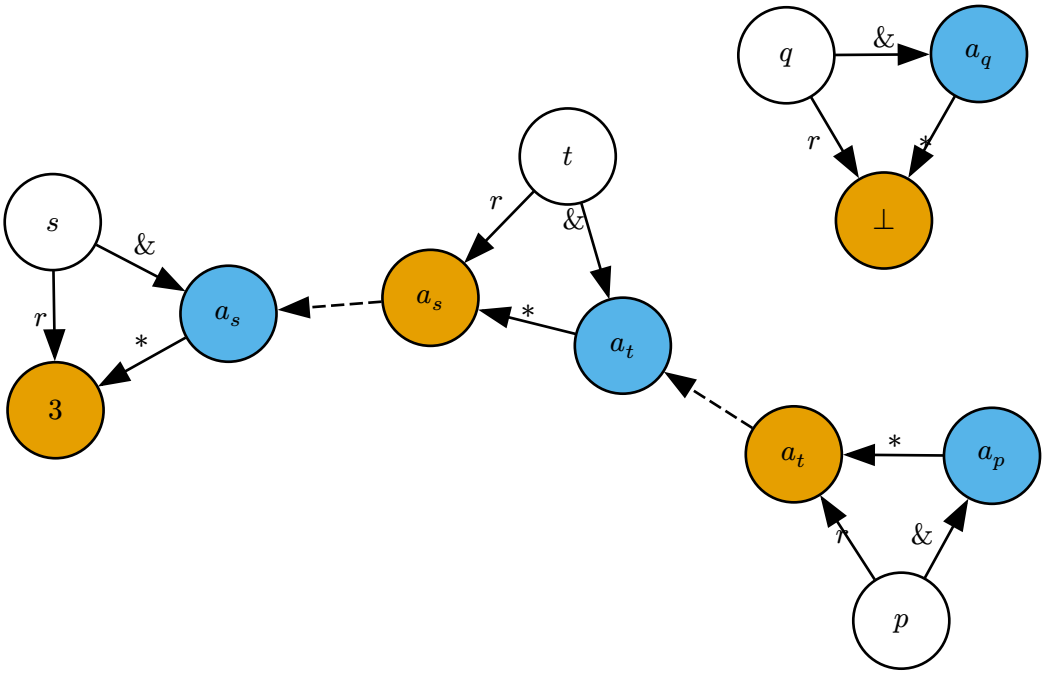
↓ int s = 3;



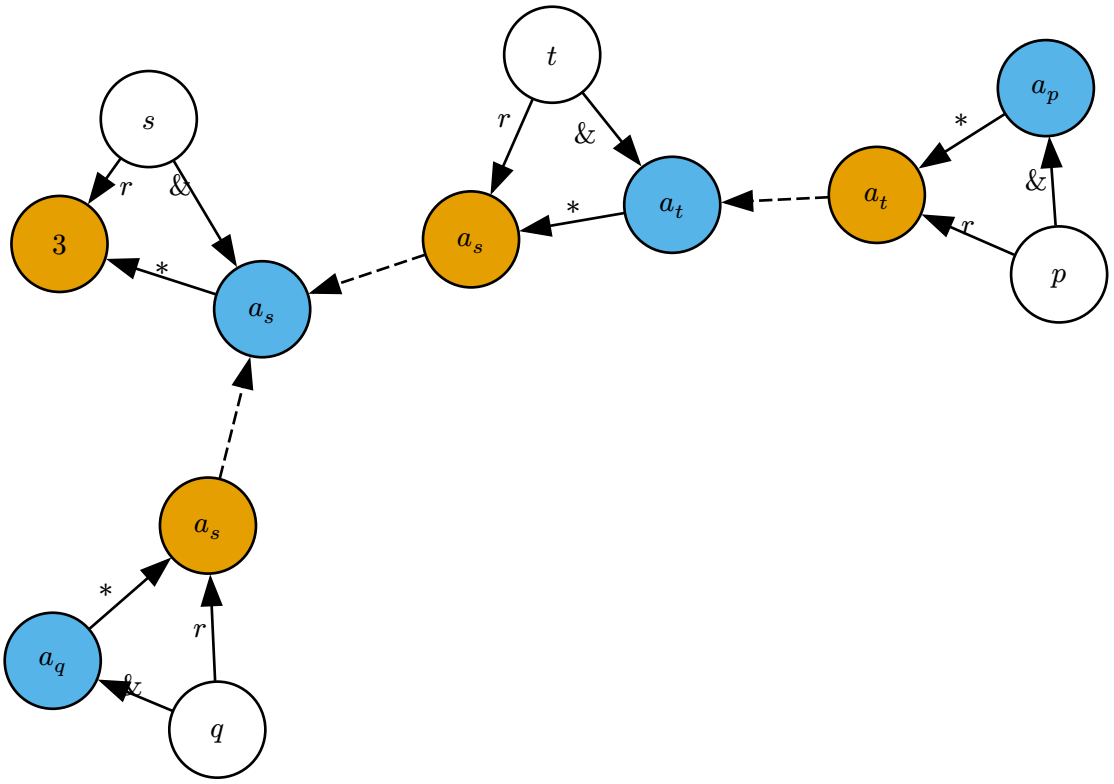
↓ int *t = &s;



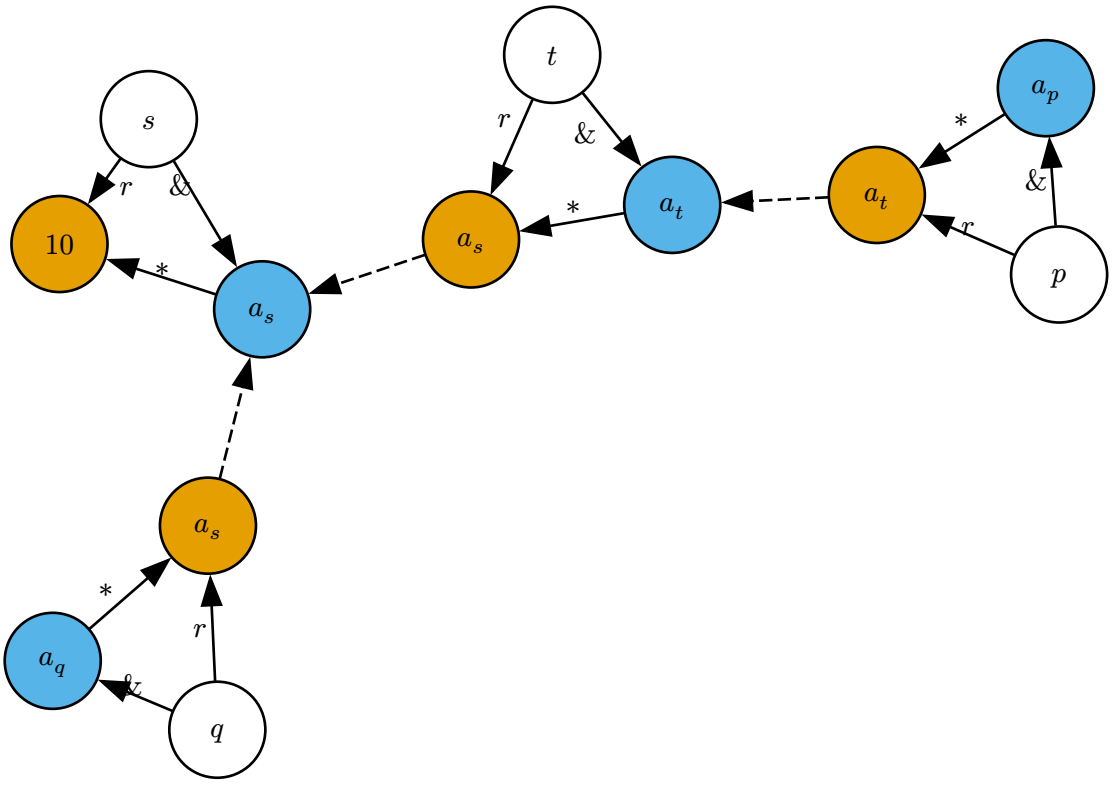
↓ int **p = &t;



↓ int *q = *p;

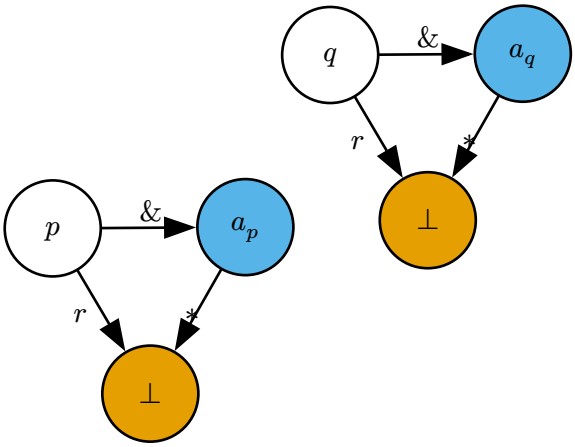


↓ *q = 10;

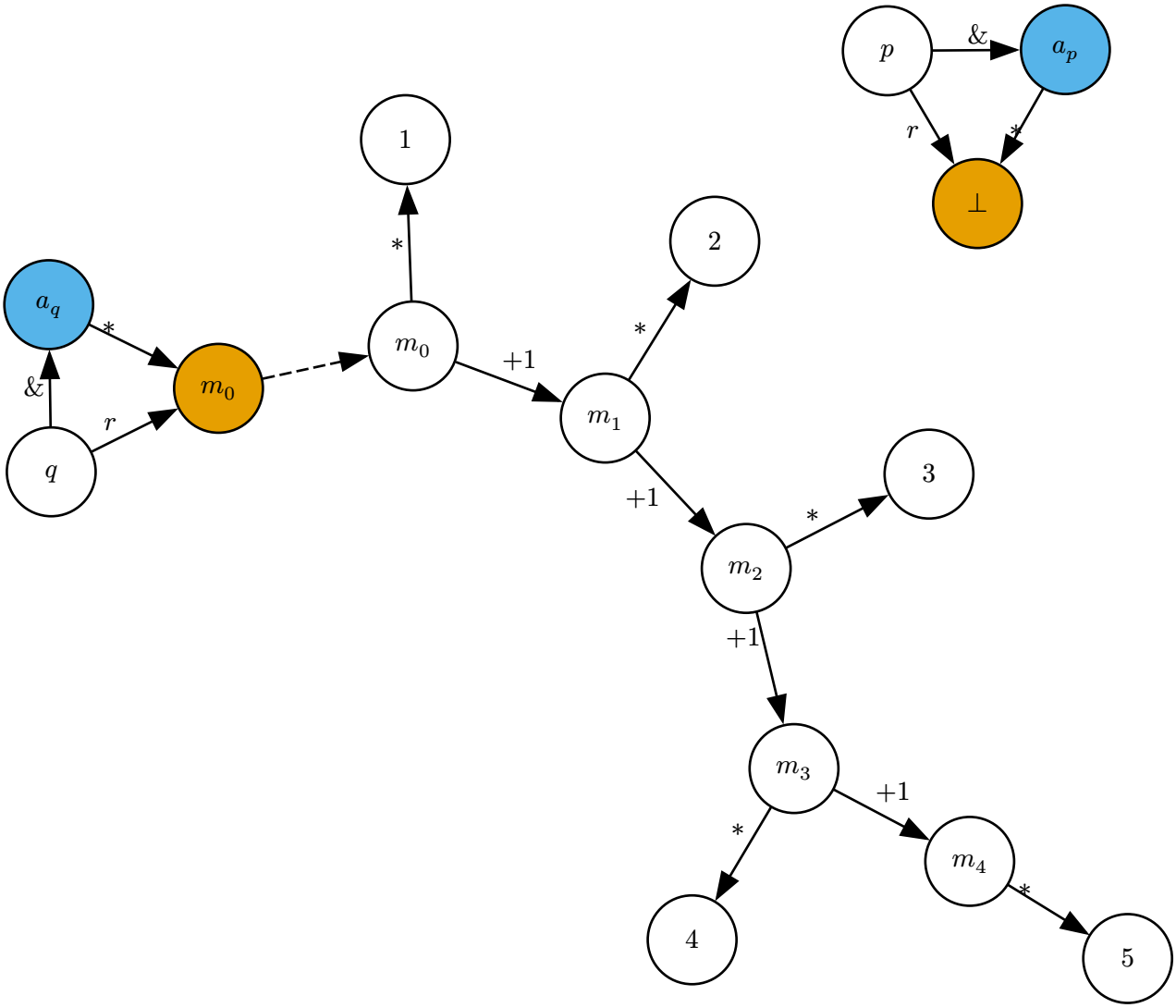


Example 5

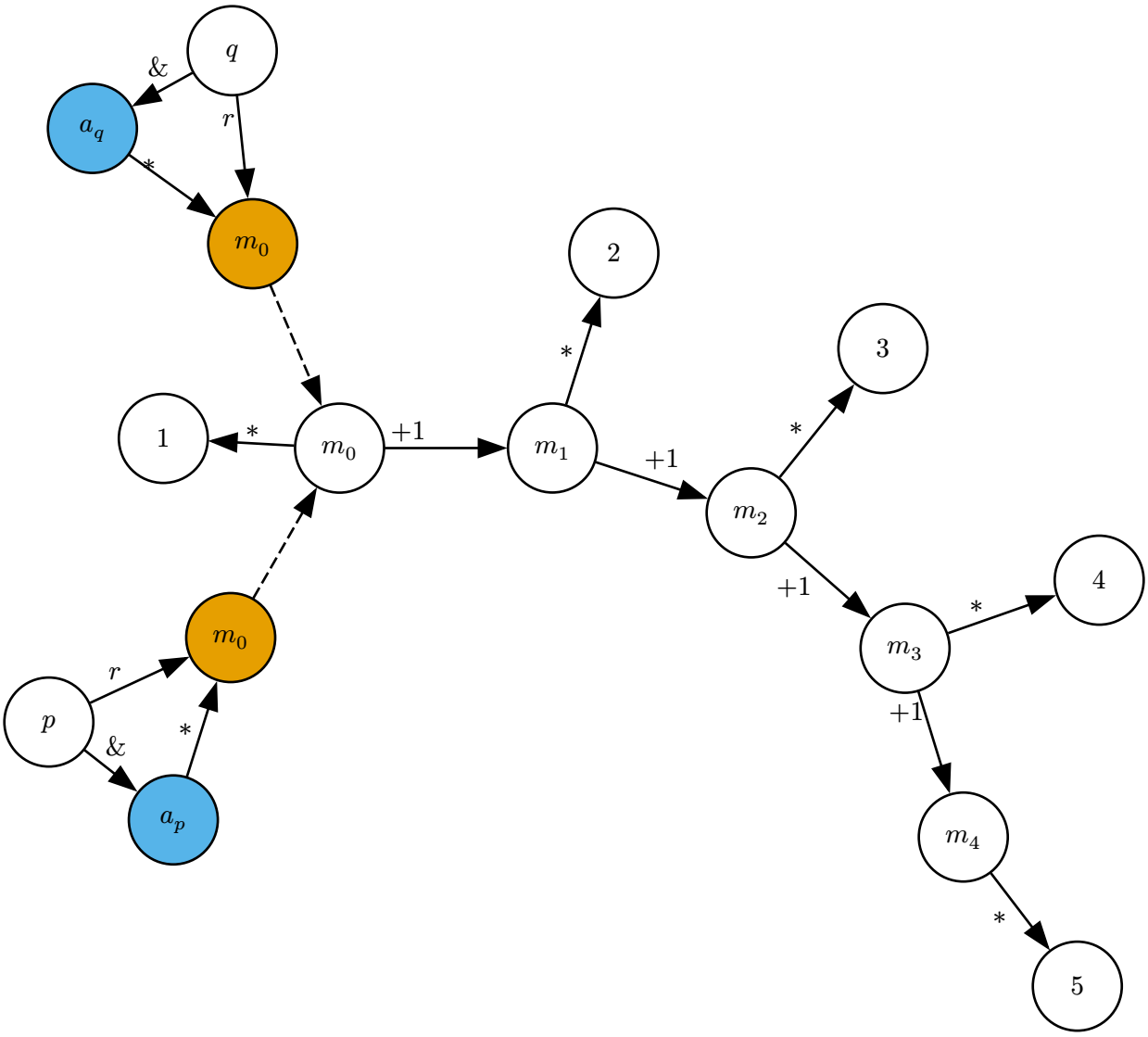
```
int q[] = {1, 2, 3, 4, 5};
int *p = q;
*(p + 2) = *(p + 4);
```



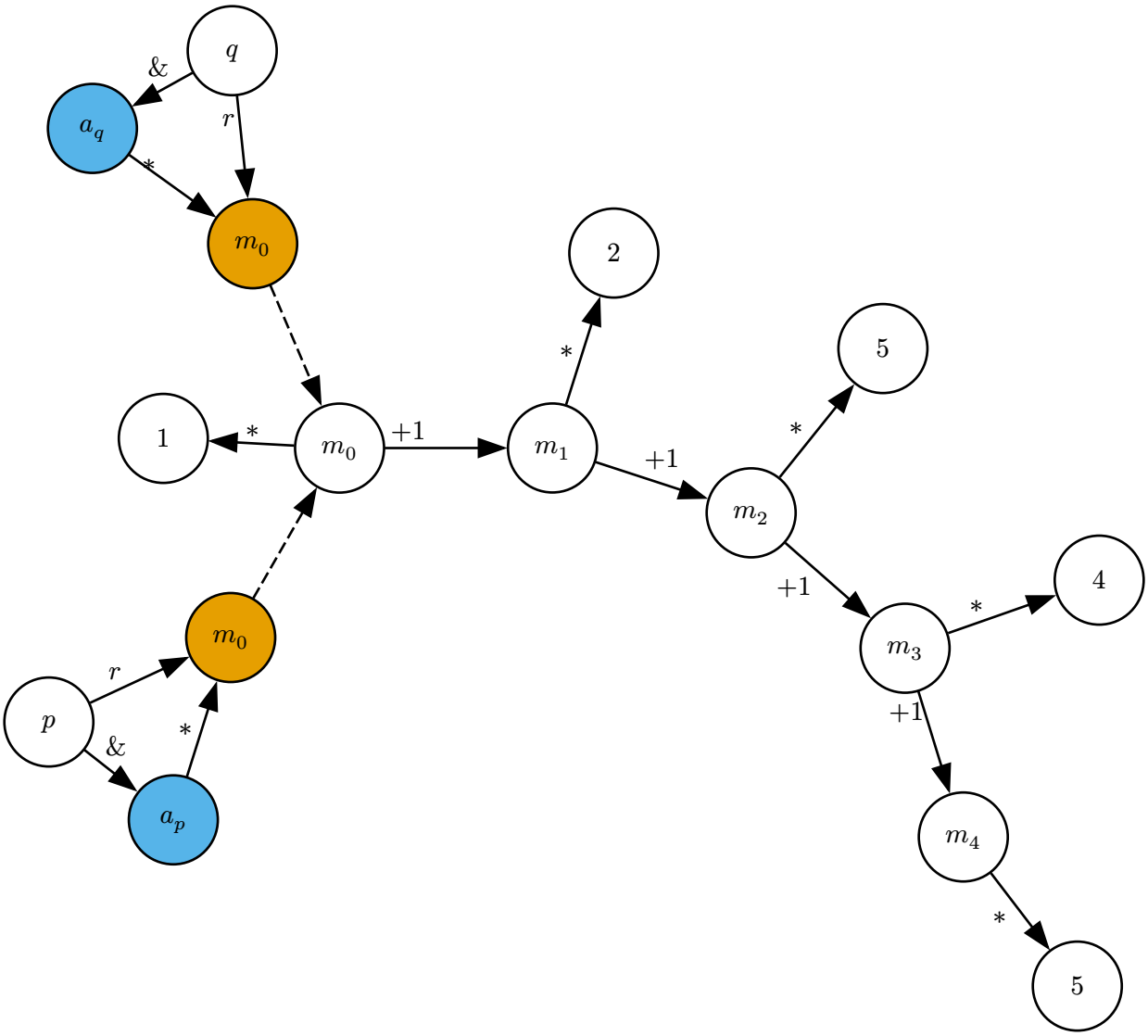
↓ int q[] = {1, 2, 3, 4, 5};



↓ int *p = q;



↓ *(p + 2) = *(p + 4);


q
&
a_q
r
*
m_0
2
*
5
1
*
m_0
+1
m_1
+1
*
m_2
+1
4
*
m_3
+1
m_0
r
p
&
*
a_p
m_4
*
5