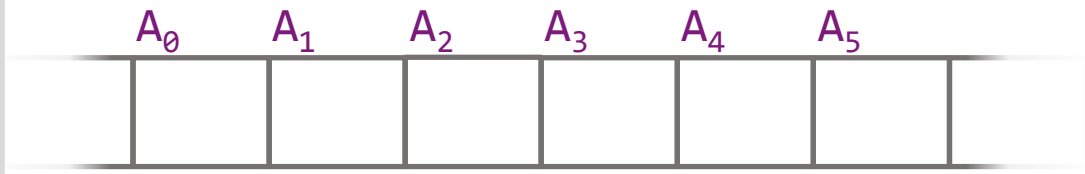
Recap: Pointers

IFMP'18, M. Schwerhoff

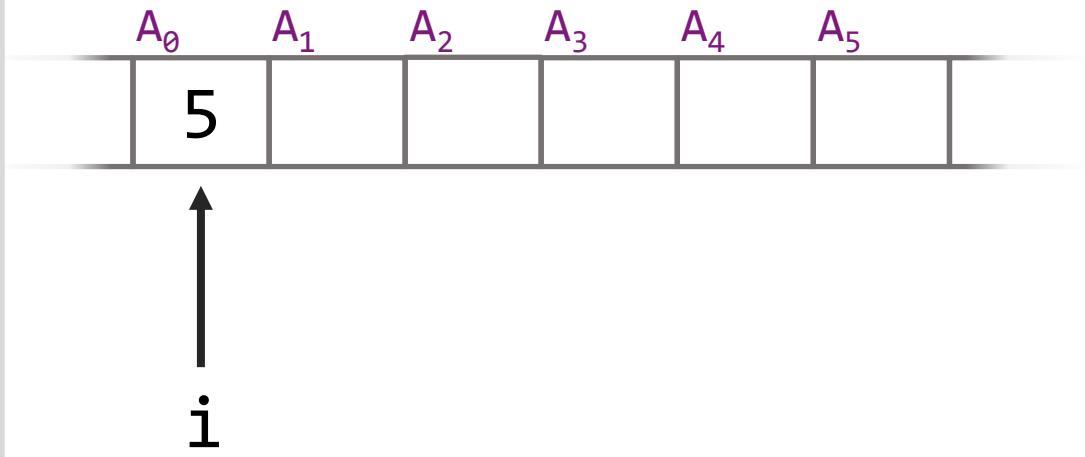
```
int* int& *p &i &*&* 🤯
```

Step-by-Step Example



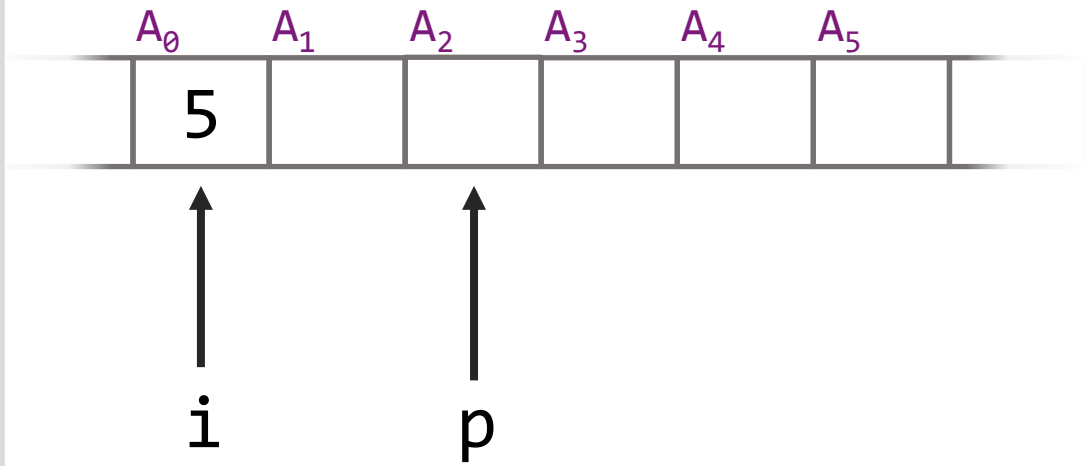
Step-by-Step Example

```
int i = 5;
```



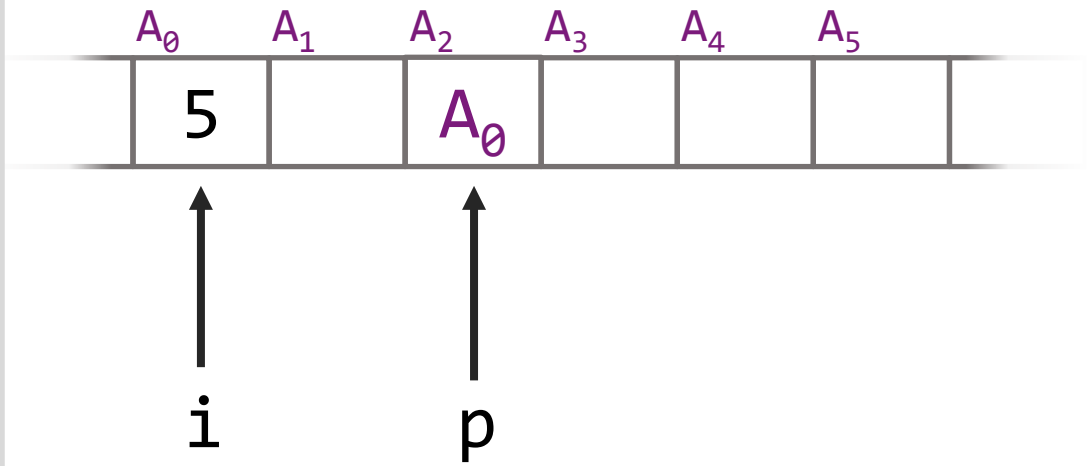
Step-by-Step Example

```
int i = 5;
int* p;
```



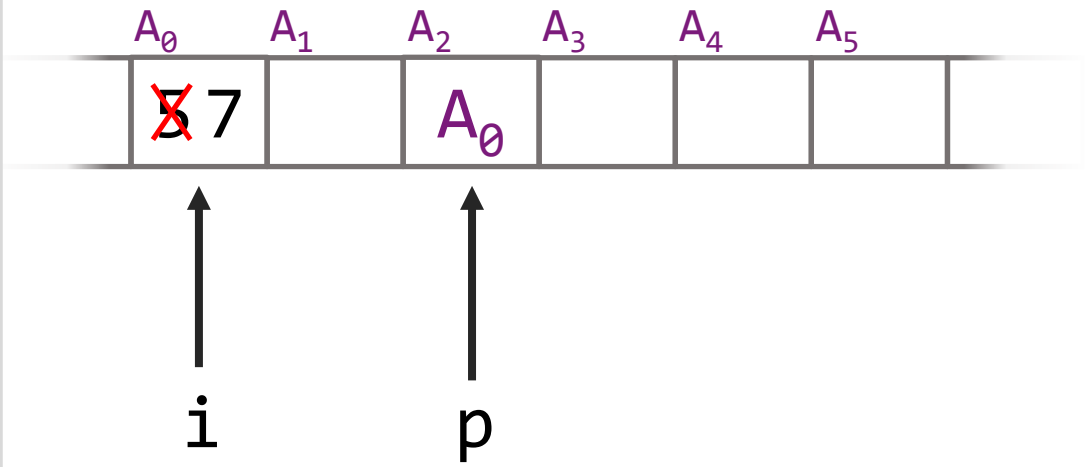
Step-by-Step Example

```
int i = 5;
int* p;
p = &i;
```



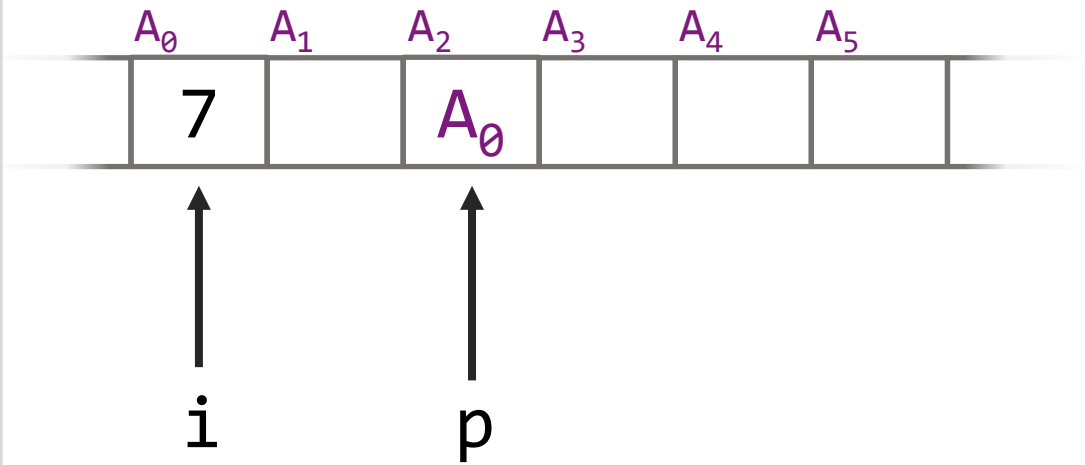
Step-by-Step Example

```
int i = 5;
int* p;
p = &i;
*p = 7;
```



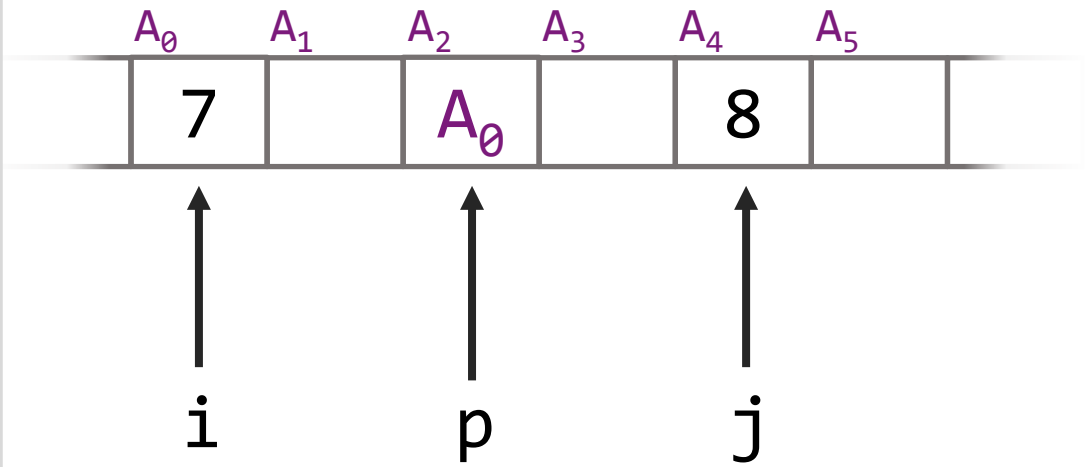
Step-by-Step Example

```
int i = 5;
int* p;
p = &i;
*p = 7;
cout << i;  // 7
```



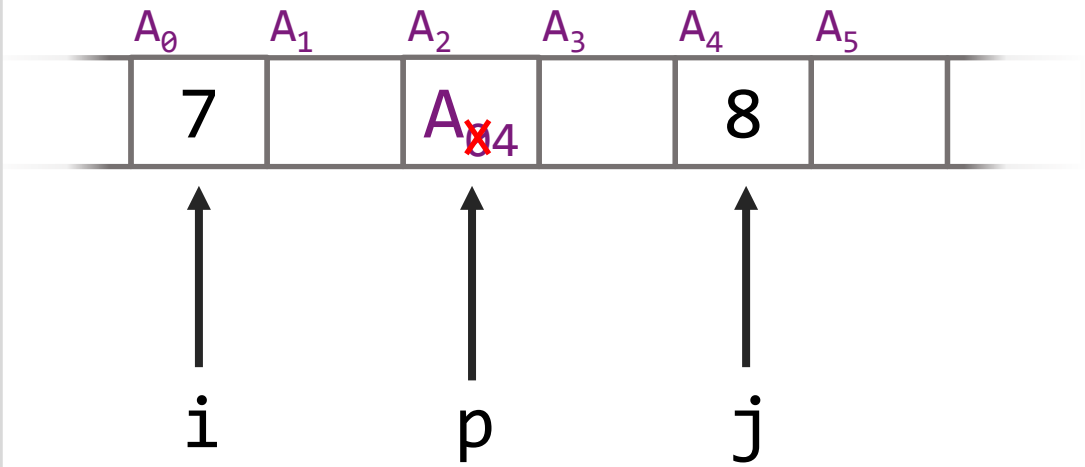
Step-by-Step Example

```
int i = 5;
int* p;
p = &i;
*p = 7;
cout << i;  // 7
int j = *p + 1;
```



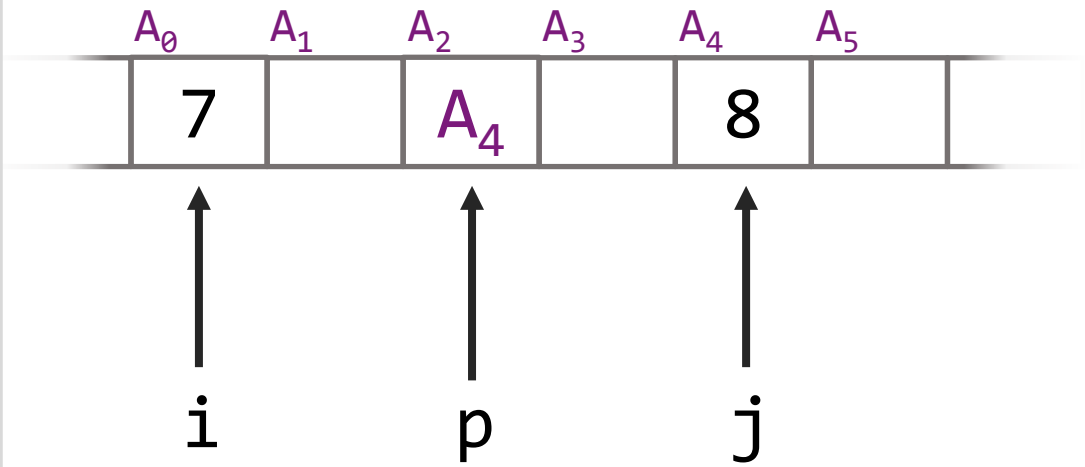
Step-by-Step Example

```
int i = 5;
int* p;
p = &i;
*p = 7;
cout << i;  // 7
int j = *p + 1;
p = &j;
```



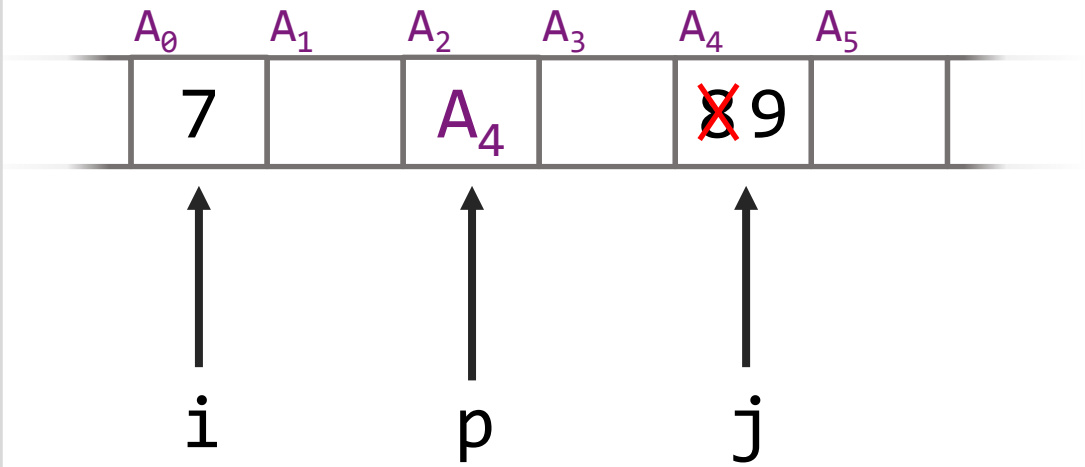
Step-by-Step Example

```
int i = 5;
int* p;
p = &i;
*p = 7;
cout << i;  // 7
int j = *p + 1;
p = &j;
cout << *p; // 8
```



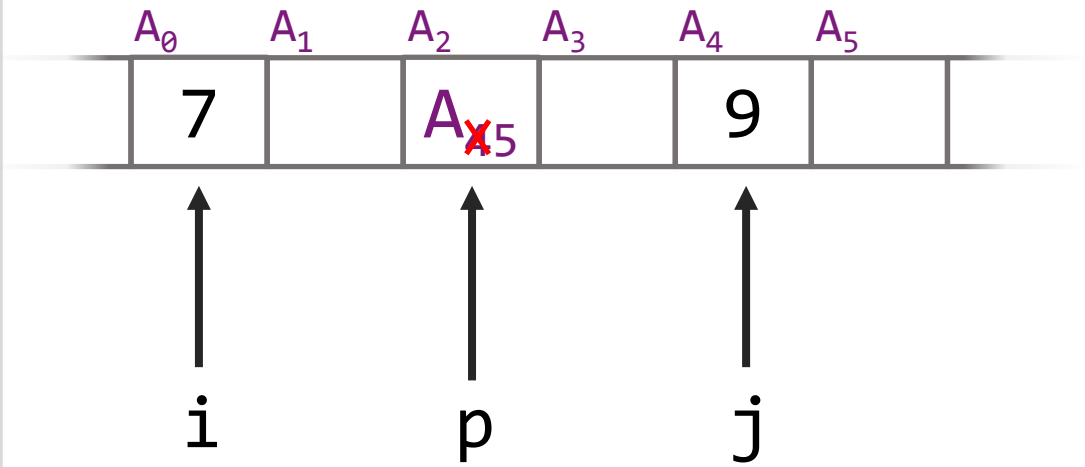
Step-by-Step Example

```
int i = 5;
int* p;
p = &i;
*p = 7;
cout << i;  // 7
int j = *p + 1;
p = &j;
cout << *p; // 8
(*p)++;
```



Step-by-Step Example

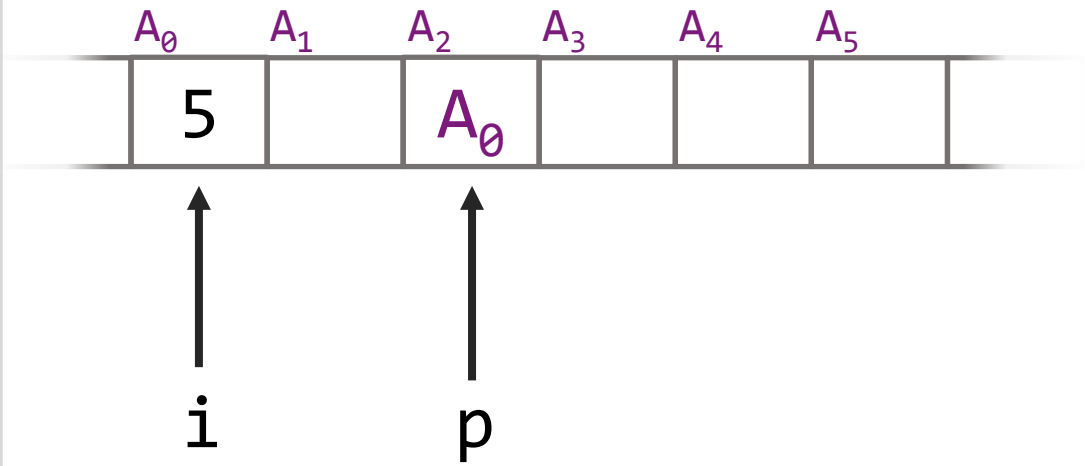
```
int i = 5;
int* p;
p = &i;
*p = 7;
cout << i;  // 7
int j = *p + 1;
p = &j;
cout << *p; // 8
(*p)++;
p++;
```



Step-by-Step Example

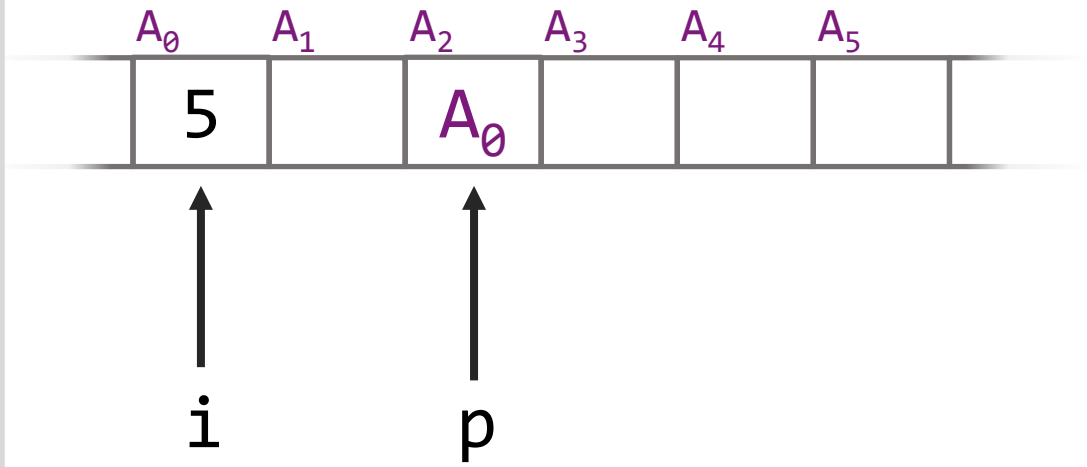
```
int i = 5;

int* p = &i;
```



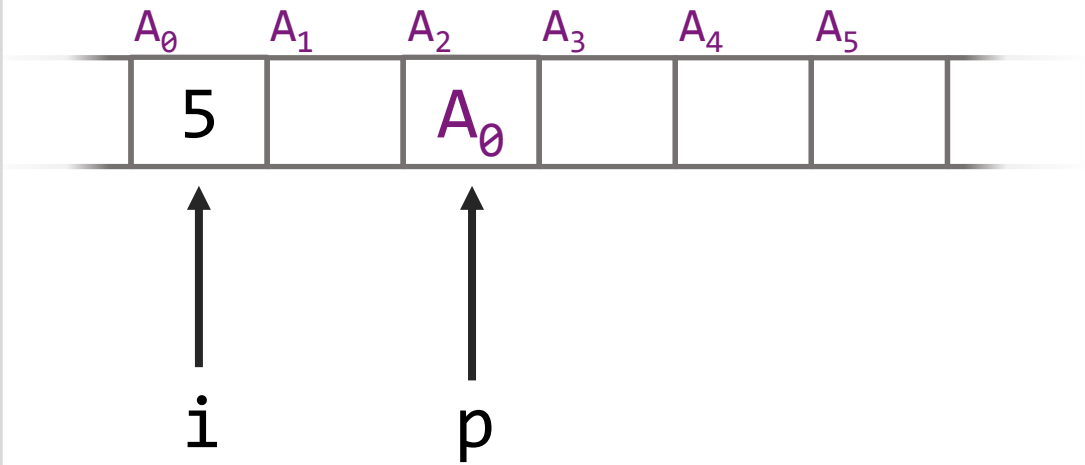
Step-by-Step Example

```
int i = 5;
int* p = &i;
cout << p; // A0
```



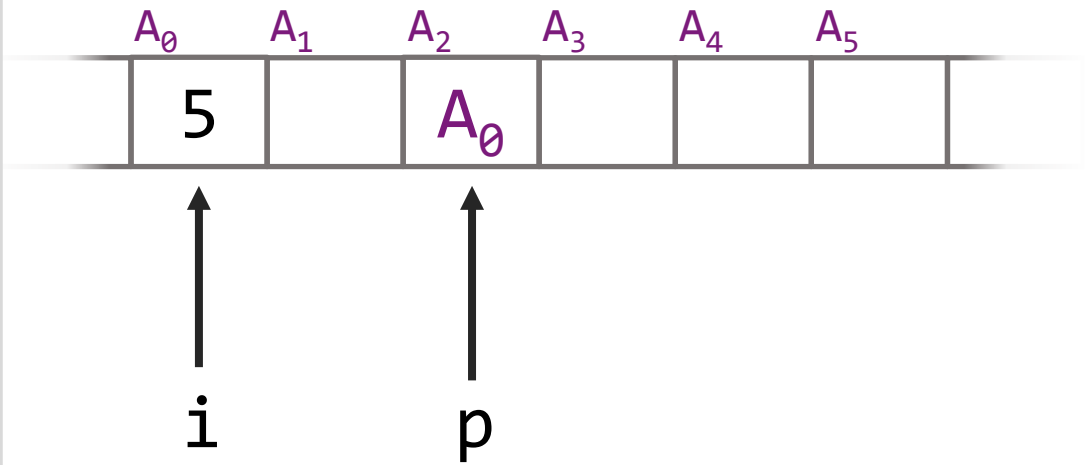
Step-by-Step Example

```
int i = 5;
int* p = &i;
cout << p; // A0
cout << *p;// 5
```



Step-by-Step Example

```
int i = 5;
int* p = &i;
cout << p; // A0
cout << *p;// 5
cout << &p;// A2
```



Recap:
Pointers vs. References

Pointers vs. References

```
int i = 5;
```

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

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

Declaration and initialisation

Pointers vs. References

```
int i = 5;
```

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

```
*p = 0; // i = 0
```

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

```
r = 0; // i = 0
```

Declaration and initialisation

Writing to underlying variable

Pointers vs. References

```
int i = 5;
```

`int* p = &i;`	`int& r = i;`	Declaration and initialisation
`*p = 0; // i = 0`	`r = 0; // i = 0`	Writing to underlying variable
`cout << *p; // 0`	`cout << r; // 0`	Reading underlying variable

Pointers vs. References

```
int i = 5;
```

`int* p = &i;`	`int& r = i;`	Declaration and initialisation
`*p = 0; // i = 0`	`r = 0; // i = 0`	Writing to underlying variable
`cout << *p; // 0`	`cout << r; // 0`	Reading underlying variable
`int* p;` `p = &i;`	~~`int& r;`~~ ~~`r = i;`~~	References must be initialised immediately

Pointers vs. References

```
int i = 5;
```

`int* p = &i;`	`int& r = i;`	Declaration and initialisation
`*p = 0; // i = 0`	`r = 0; // i = 0`	Writing to underlying variable
`cout << *p; // 0`	`cout << r; // 0`	Reading underlying variable
`int* p;` `p = &i;`	~~`int& r;`~~ ~~`r = i;`~~	References must be initialised immediately
`int* p = &i;` `p = &j;`	`int& r = i;` ~~`r = j;`~~	References *themselves* cannot be changed (“redirected”)

Pointers vs. References

```
int i = 5;
```

`int* p = &i;`	`int& r = i;`	Declaration and initialisation
`*p = 0; // i = 0`	`r = 0; // i = 0`	Writing to underlying variable
`cout << *p; // 0`	`cout << r; // 0`	Reading underlying variable
`int* p;` `p = &i;`	~~`int& r;`~~ ~~`r = i;`~~	References must be initialised immediately
`int* p = &i;` `p = &j;`	`int& r = i;` ~~`r = j;`~~	References *themselves* cannot be changed (are always `const`)
`int* p = &i;` `cout << &p;`	`int& r = i;` ~~`cout << &r;`~~	References don't have addresses (they are just aliases of sth. else)

Why Pointers and References?

Why Pointers *and* References?

Historical language development:

- C: 1969; has pointers
- C++: 1983
  - Inherited pointers from C (backwards compatibility)
  - Added references to support *operator overloading*

Why Pointers *and* References?

Historical language development:

- C: 1969; has pointers
- C++: 1983
  - Inherited pointers from C (backwards compatibility)
  - Added references to support *operator overloading*

Rule of thumb: prefer references over pointers, if possible

- References are more restricted → harder to make mistakes
- Reference syntax is nicer (`r` vs `*p`) → easier to read

Why Pointers *and* References?

When do we *need* pointers?

- Dynamic memory allocation (`new`)
- Fast(er) *iteration over data*: e.g. arrays (last week) and containers (today)

Why Pointers *and* References?

When do we *need* pointers?

- Dynamic memory allocation (`new`)
- Fast(er) *iteration over data*: e.g. arrays (last week) and containers (today)
- *Dynamic data structures* that *change* over time, such as lists, trees or graphs