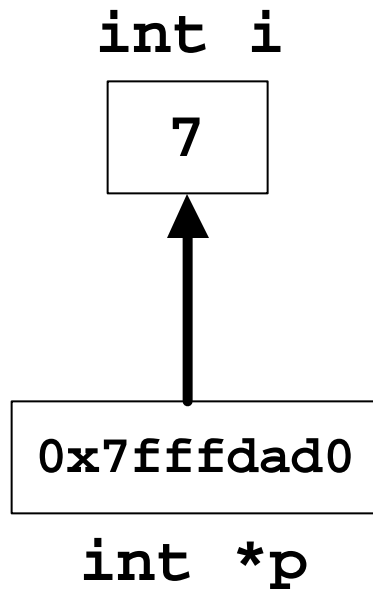


Motivation and Overview

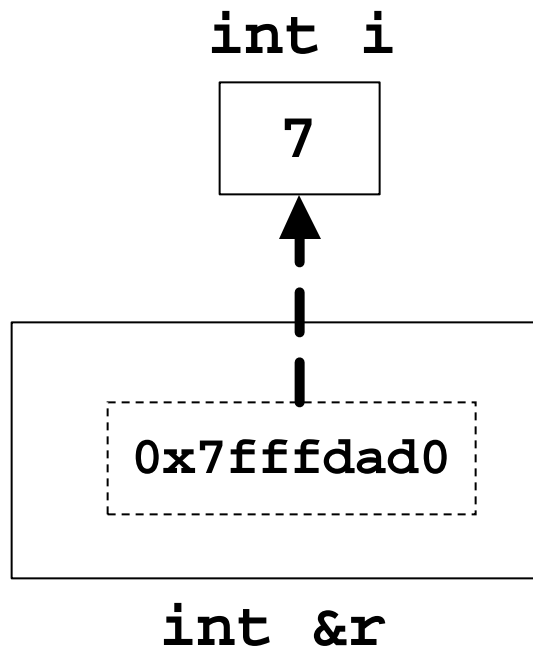
- Often need to *refer* to another object
 - Without making a copy of the object itself
- Two ways to do this
 - Indirectly, via a *pointer*
 - Gives the address (in memory) of the object
 - Requires the user to do extra work: dereferencing
 - Directly, via a *reference*
 - Acts as an alias for the object
 - User interacts with reference as if it were the object itself

What's a Pointer?



- A variable holding an address
 - Of what it “points to” in memory
- Can be untyped
 - `void * v; // points to anything`
- However, usually they’re typed
 - Checked by compiler
 - Can only be assigned addresses of variables of type to which it can point
 - `int * p; // only points to int`
- Can point to garbage or nothing
 - When created: `int *p;`
 - `p = NULL; // points to nothing`

What's a Reference?



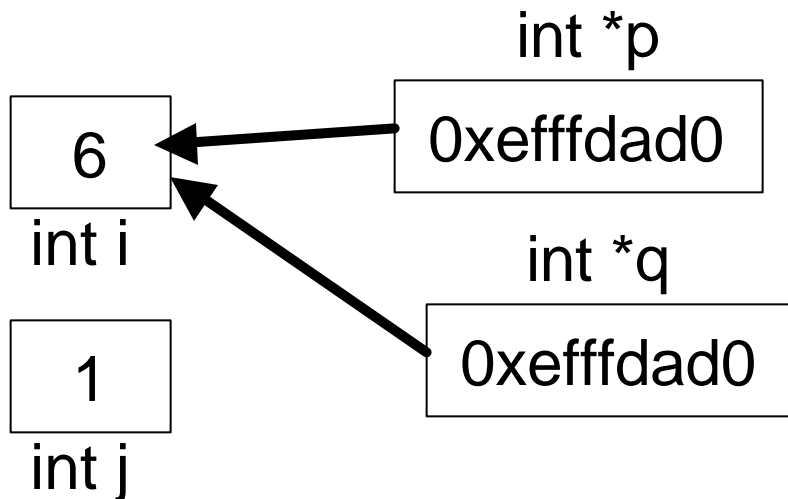
- A variable holding an address
 - Of what it “refers to” in memory
- But with a nicer interface
 - An alias to the object
 - Hides indirection from programmer
- Must be typed
 - Checked by compiler
 - Again can only refer to the type to which it can point
 - `int &r; // only refers to int`
- Must always refer to something

Untangling Operator Syntax

Symbol	Used in a declaration	Used in a definition
unary & (ampersand)	reference <ul style="list-style-type: none">• <code>int i;</code>• <code>int &r = i;</code>	address-of <ul style="list-style-type: none">• <code>p = &i;</code>
unary * (star)	pointer <ul style="list-style-type: none">• <code>int * p;</code>	dereference <ul style="list-style-type: none">• <code>*p = 7;</code>
-> (arrow)		member access via pointer <ul style="list-style-type: none">• <code>cp->add(3);</code>
• (dot)		member access via reference or object <ul style="list-style-type: none">• <code>c.add(3);</code>

Aliasing and Pointers

```
int main(int argc, char **argv) {  
    int i = 0;  
    int j = 1;  
    int *p = &i;  
    int *q = &i;  
    *q = 6;  
    // i is now 6, j is still 1  
}
```



- Distinct variables alias *different* memory locations
 - E.g., `i` and `j`
- An object and all the pointers to it (when they're *dereferenced*) alias the same location
 - E.g., `i`, `*p`, and `*q`
- Assigning a new value to `i`, `*p` or `*q` changes value seen through the others
- But does not change value seen through `j`

Const Pointers

```
int main (int argc, char **argv) {
    const int i = 0;
    int j = 1;
    int k = 2;

    // pointer to int
    int *w = &j;

    // const pointer to int
    int *const x = &j;

    // pointer to const int
    const int *y = &i;

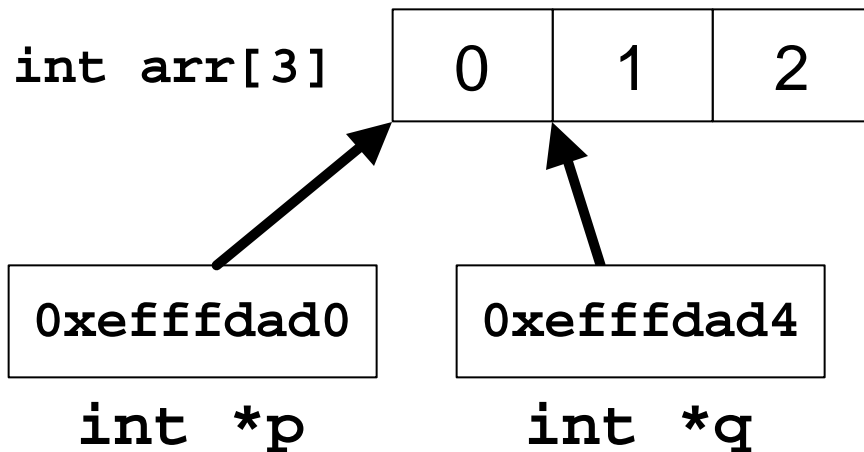
    // const pointer to const int
    const int const *z = &j;
}
```

- Make promises via the **const** keyword in pointer declaration:
 - not to change where the pointer points
 - not to change what it points to
- Read declarations right to left
- can change
 - **w** and what it points to
 - what **x** points to but not **x**
 - **y** but not what it points to
 - neither **z** nor what it points to
- A pointer to const cannot point to a non-const variable
 - **w** and **x** can't point to **i**

Pointers and Arrays

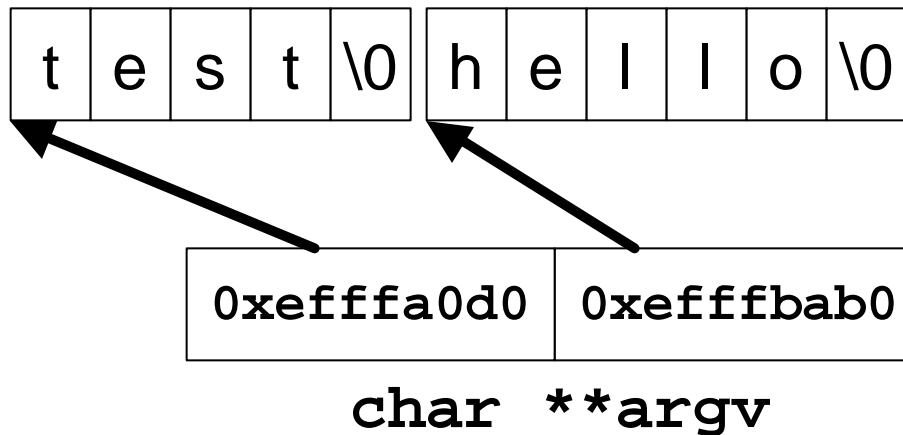
```
int main (int argc, char **argv) {  
    int arr[3] = {0, 1, 2};  
  
    int * p = &arr[0];  
    int * q = arr;  
    // p, q now point to same place  
  
    ++q; // now p points to arr[1]  
}
```

- An array holds *contiguous* memory locations
- Array variable behaves like a const pointer
 - *i.e.*, `int * const arr;`
- Can initialize other pointers to the start of the array
 - Using array name
 - Using address of 0th element
- Pointer arithmetic
 - Adding number `n` to a pointer moves `n` of the type to which it points
 - *i.e.*, `n` array positions
 - *E.g.*, value in `q` increased by `sizeof(int)` by `p++`



Arrays of (and Pointers to) Pointers

```
int main(int argc, char **argv) {  
    for (int i = 0; i < argc; ++i)  
        cout << argv[i] << endl;  
  
    return 0;  
}
```



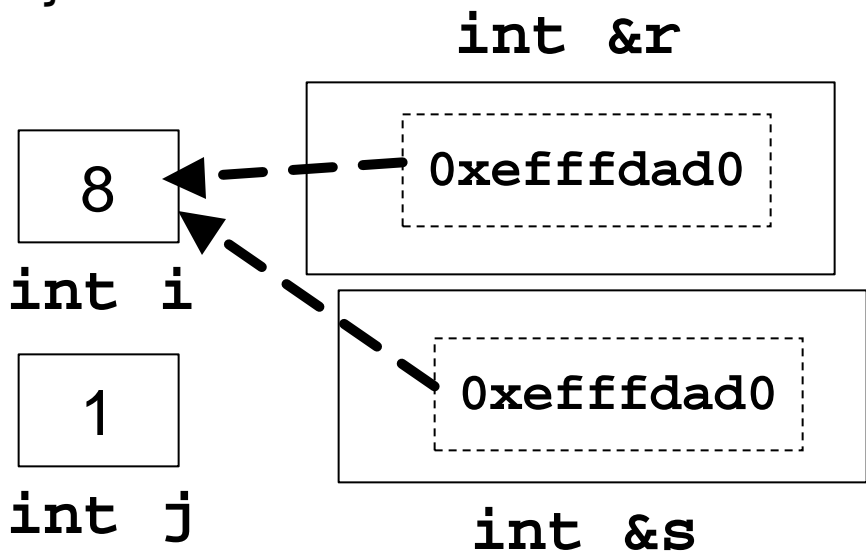
```
int argc    2
```

- Can have pointers to pointers
- Can also have an array of pointers (like a const pointer to a pointer type)
- E.g., `argv` parameter in `main`
 - Array of pointers to character strings
 - Could also declare as a pointer to the first pointer
 - Array dimension is not specified
 - Instead a special argument (`argc`) holds array size
- By convention, character strings are zero terminated
 - Special char is `'\0'` not `'0'`

Aliasing and References

```
int main(int argc, char **argv) {  
    int i = 0;  
    int j = 1;  
    int &r = i;  
    int &s = i;  
    r = 8;  
    // i is now 8, j is still 1  
}
```

- An object and all the references to it alias the same location
 - *E.g.*, i, r, and s
- Assigning a new value to i, r or s changes value seen through the others
- But does not change value seen through j



Const References

```
int main (int argc, char **argv) {
    const int i = 0;
    int j = 1;

    // r can't refer to i
    int &r = j;

    // this is ok, though
    const int &s = i;
    const int &t = j;
}
```

- Remember: references must refer to something
 - Can't be `NULL`
- Also, once initialized, they cannot be changed
 - E.g., can't redirect `t` to `i`
- Const on a reference
 - A promise not to change what's aliased
 - E.g., can't use `t` to change `j`
- Can't have a non-const reference alias a const variable
 - Reverse is OK

Parameter Passing

- By value

```
int main (int argc, char **argv) {  
    int h = -1;  
    int i = 0;  
    int j = 1;  
    int k = 2;  
    return func(h, i, j, &k);  
}
```

```
int func(int a, const int &b,  
        int &c, int *d) {  
    ++a;  
    c = b;  
    *d = c  
    ++d;  
  
    return 0;  
}
```

– Makes a copy i.e., of **h** into local variable **a**
– **++a** does not change **h**

- By reference

– Alias for passed variable
– **c = b** changes **j**
– can't change **b** (or **i**): **const**

- Can pass address by value

– And then use address value to change what it points to
– ***d = c** changes **k**
– **++d** changes local pointer

References to Pointers

```
int main (int argc, char **argv) {  
    int j = 1;  
    int &r = j;    // r aliases j  
    int *p = &r;  // p really  
                  // points to j  
    int * &t = p; // t aliases p  
}
```

- Can't have a pointer to a reference
 - But can point to what the reference aliases
- Address-of operator on a reference to a variable
 - Gives address of variable
 - ... not of reference itself
- Reference to a pointer
 - An alias for the pointer
 - ... not for what it points to
 - Useful to pass a pointer to code that may change it

