Pointers (part 1)

CSC 230 : C and Software Tools

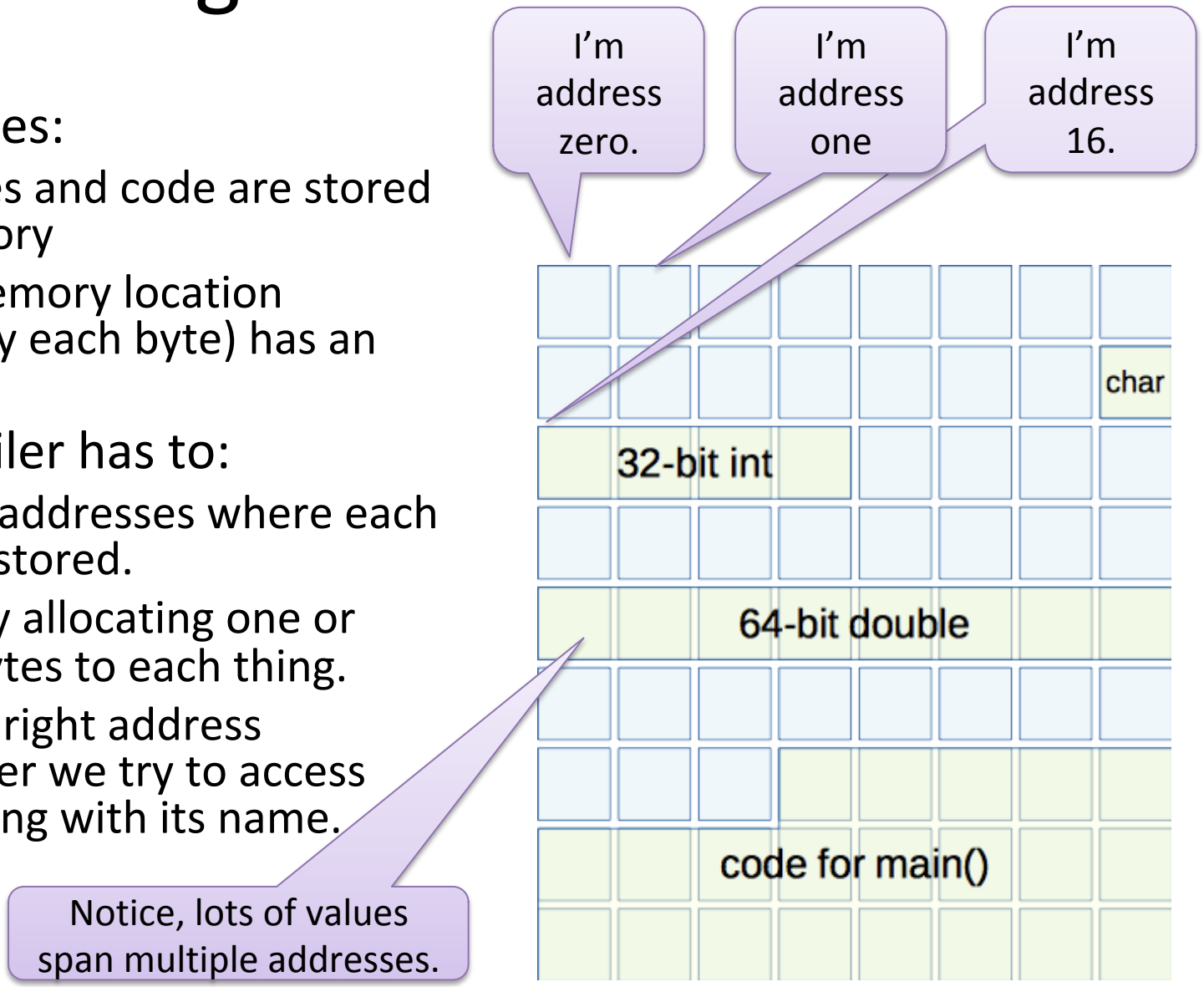
NC State Department of Computer Science

Topics for Today

- Memory and addresses
- Pointer syntax
  - One new type operator
  - Two new operators
- Meet NULL
- Pass-by-reference
- Good and bad pointer syntax
- Reading types
- Your friend : const

Thinking about Addresses

- No surprises:
  - Variables and code are stored in memory
  - Each memory location (typically each byte) has an address
- The compiler has to:
  - Choose addresses where each thing is stored.
  - Typically allocating one or more bytes to each thing.
  - Use the right address whenever we try to access something with its name.



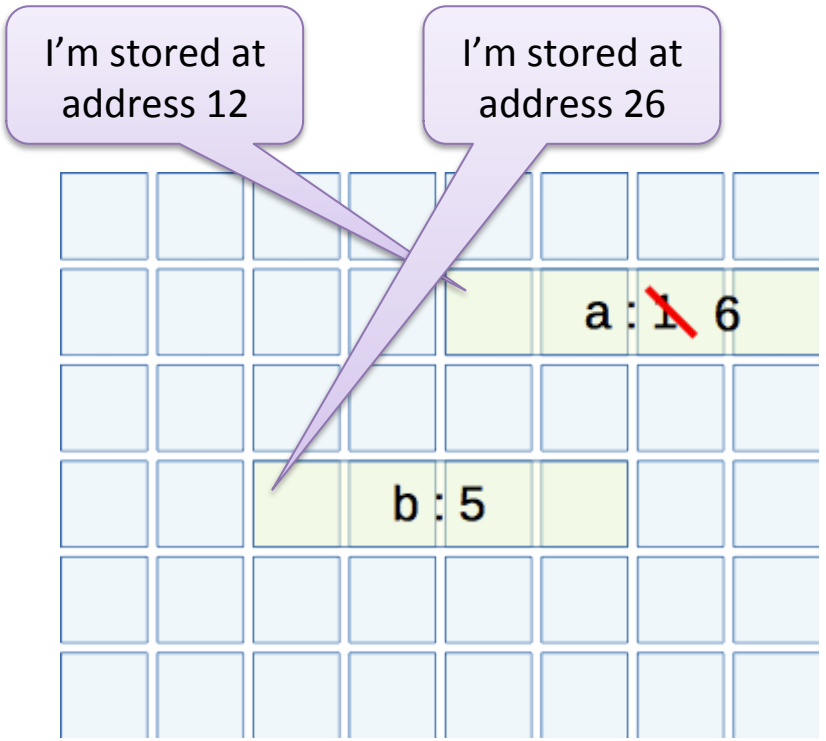
Thinking about Addresses

- When we say :

```
a = b + 1
```

- The compiler writes code like :

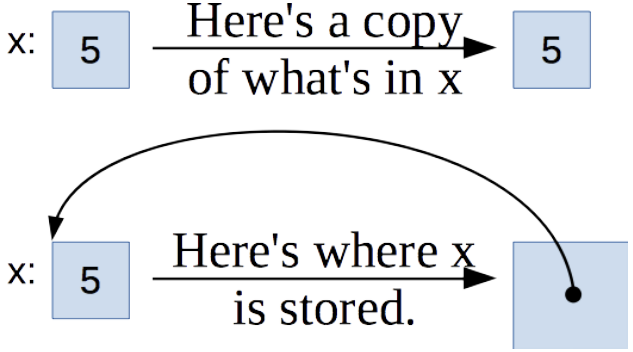
```
Get the int at address 26
Add 1
Store the result at address 12
```



- Normally, this is all hidden from us … but the compiler will let you work with addresses **if you want to.**

Why Would You Want To?

- There are three general reasons why we need to work with addresses:
  - Knowing where something is stored will let you read it **and** change it.



  - Knowing where something is stored will let you access nearby values in memory.
  - Dynamic memory allocation requires us to work with addresses.

That's how arrays work.

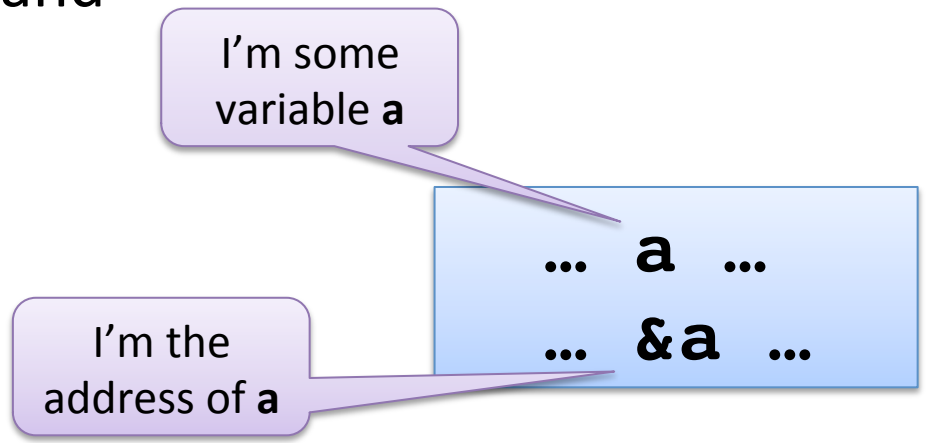
Indirection

- Pointers are a mechanism for *indirection*
- Instead of working directly with values, we work with the locations where values are stored.

Where “work with” means store, pass to functions, etc.

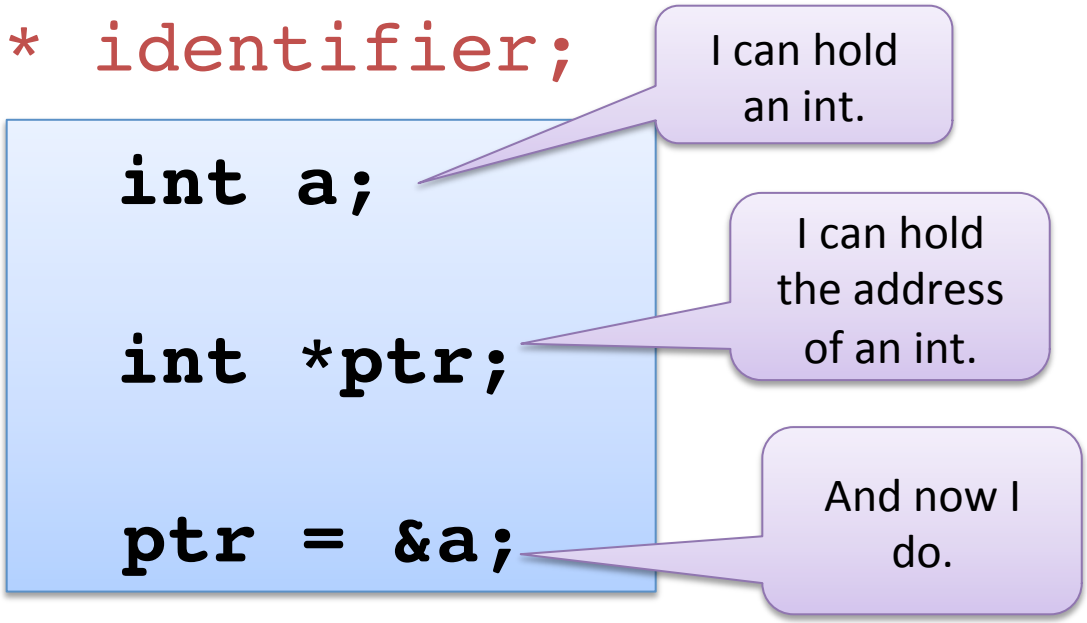
Working with Addresses

- We're going to see two new operators:
  - One for asking where things are stored in memory
  - One for accessing what's stored at particular memory locations
- The address-of operator : &
  - It's a unary prefix operator that evaluates to the address of its operand



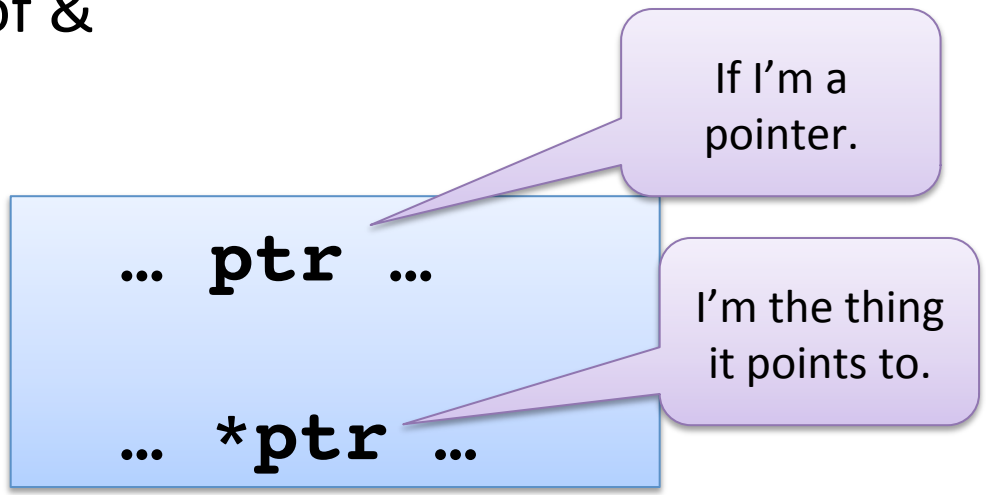
Storing Addresses

- We have special kinds of variables for storing addresses, pointer variables.
- The * symbol can be used as a *type operator*. It lets us create new types.
  - Syntax: `type * identifier;`



Using a Pointer

- Given a pointer, we can access the value it points to.
- We have the *dereference operator*: *
  - I know, it looks just like the pointer type operator, but we use this one differently.
  - It's a unary prefix operator
  - It's the inverse of &



* and &

- * and & work together
  - but they're inverses of each other.
- **&** says "Give me the address of this thing"

```
pointer = &thing;
```

- * says "give me the thing at this address"
  - dereference will give you the thing a pointer points to.

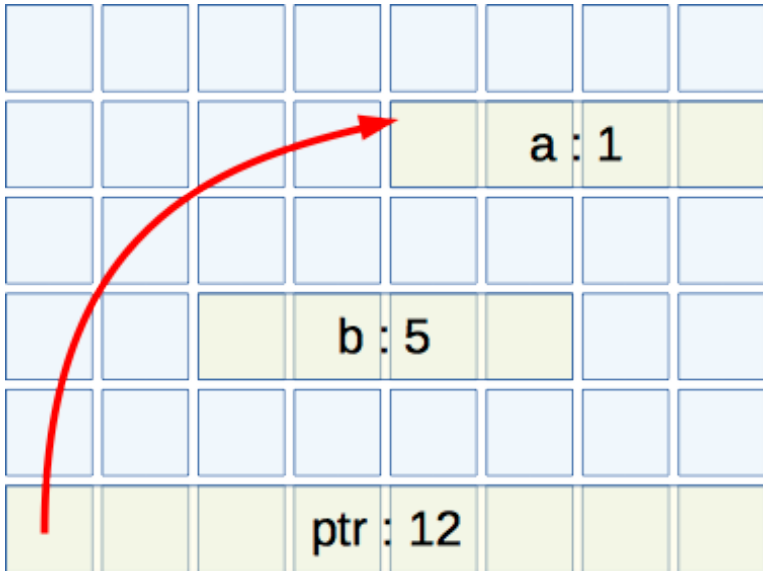
```
x = *pointer;
```

  - Or, it will let you change it change it.

```
*pointer = y;
```

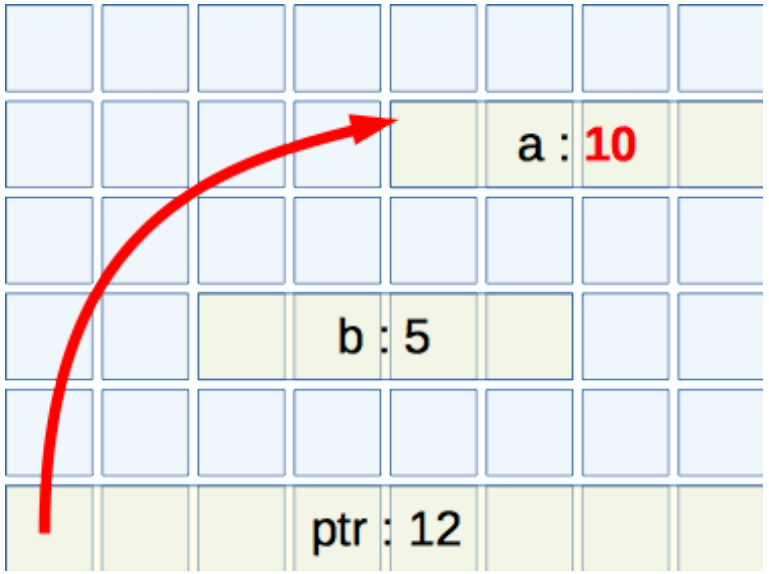
Pointer Illustration

```
int a = 1;
int b = 5;

int *ptr = &a;
```



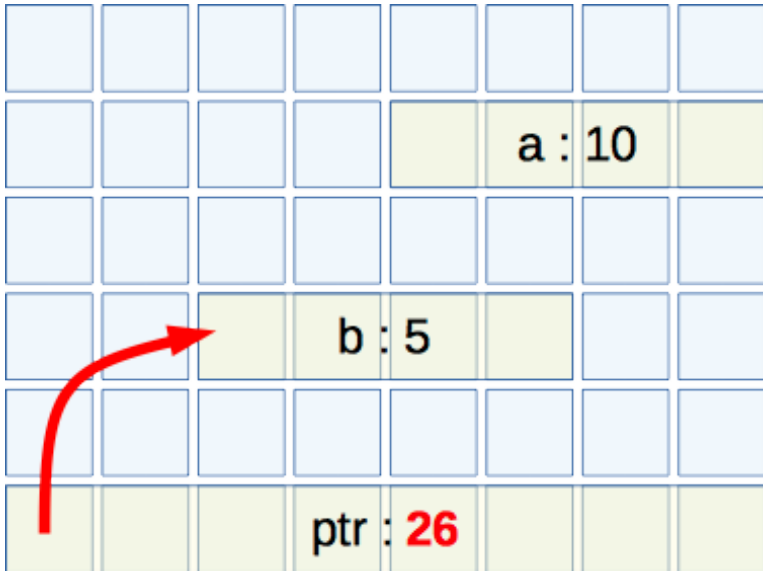
Pointer Illustration

```
int a = 1;
int b = 5;

int *ptr = &a;
*ptr = 10;
```



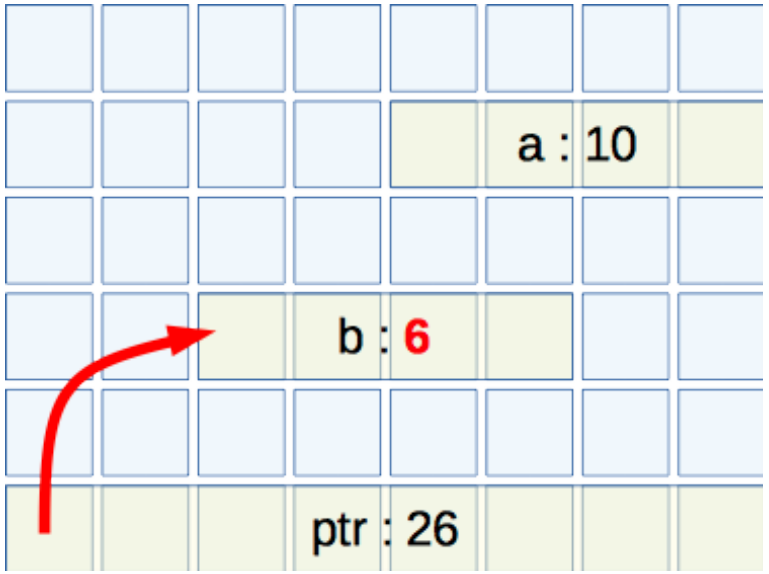
Pointer Illustration

```
int a = 1;
int b = 5;

int *ptr = &a;
*ptr = 10;
ptr = &b;
```



Pointer Illustration

```
int a = 1;
int b = 5;

int *ptr = &a;
*ptr = 10;
ptr = &b;
++*ptr;
```



Try it out Yourself

```
int a = 5;
int b = 10;
int *pa = &a;
int *pb = &b;

*pa += 1;
*pb += *pa;
pb = pa;
*pb += 4;
```

What do each of these lines change?

All Types Can Have Pointers

```
char c = 'x';
char *cp = &c;
```

I'm a pointer to a char.

```
float f = 22.7;
float *fp = &f;
```

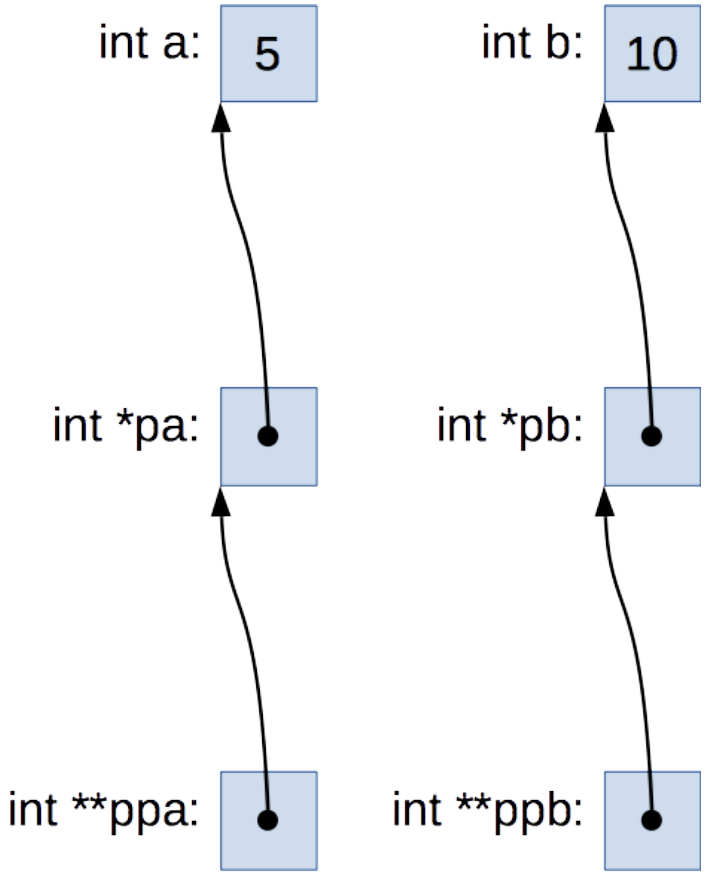
I'm a pointer to a float.

```
short s = -192;
short *sp = &s;
```

I'm a pointer to a short.

Pointers to … Pointers

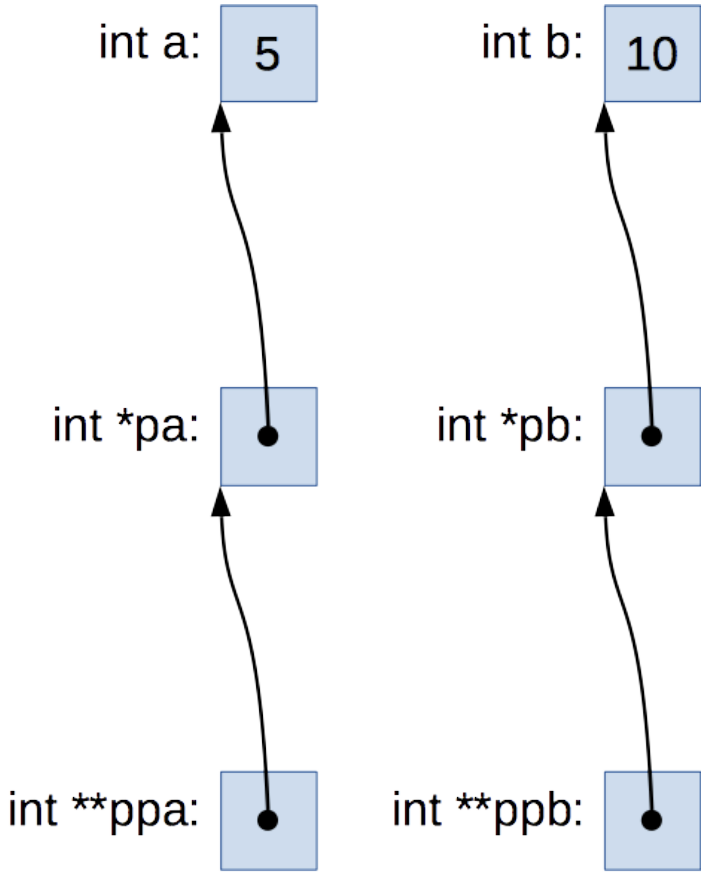
```
int a = 5;
int b = 10;
int *pa = &a;
int *pb = &b;
int **ppa = &pa;
int **ppb = &pb;
```



Pointers to … Pointers

```
int a = 5;
int b = 10;
int *pa = &a;
int *pb = &b;
int **ppa = &pa;
int **ppb = &pb;

**ppa = 15;
```

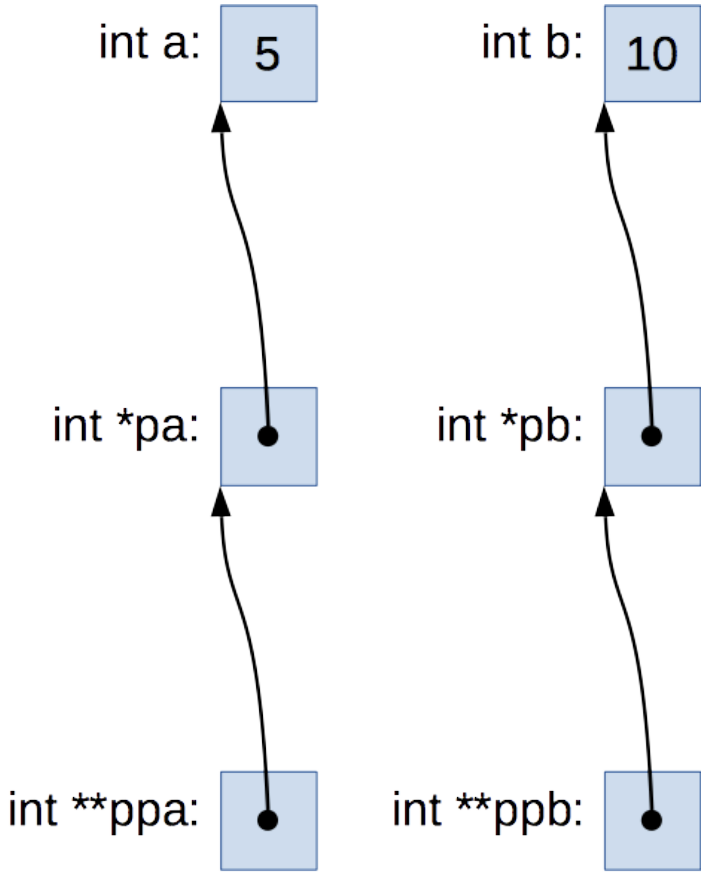
What will this change?



Pointers to … Pointers

```
int a = 5;
int b = 10;
int *pa = &a;
int *pb = &b;
int **ppa = &pa;
int **ppb = &pb;

*ppa = &b;
**ppa += 1;
```

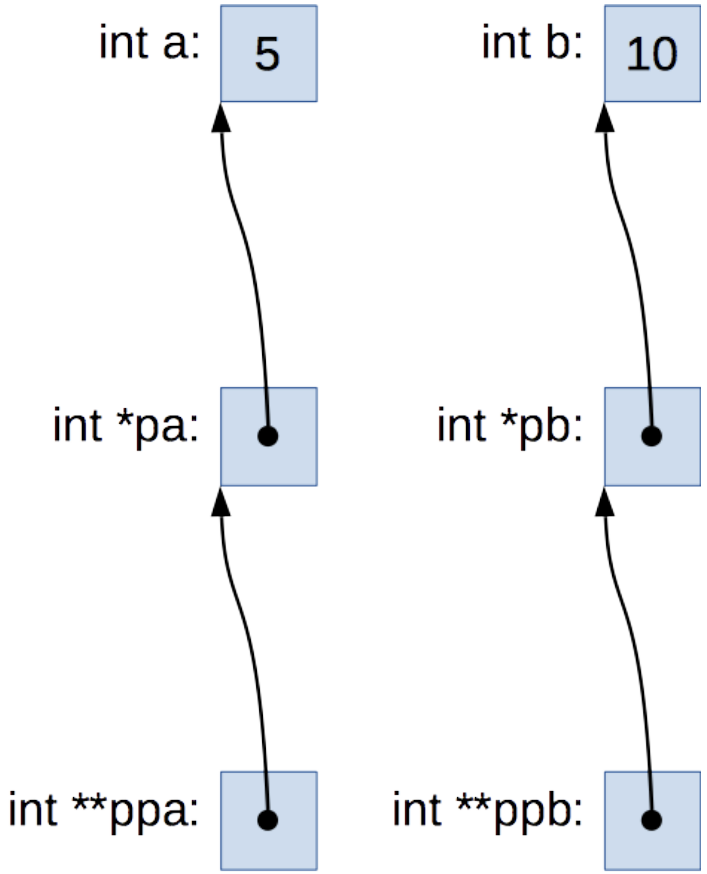
How about this?



Pointers to … Pointers

```
int a = 5;
int b = 10;
int *pa = &a;
int *pb = &b;
int **ppa = &pa;
int **ppb = &pb;

ppa = &pb;
**ppa -= 1;
```

Or this?



A Pointer to Nowhere

- C defines the constant, NULL
  - It's a pointer to nothing
  - Like null in Java
  - Defined in **stddef.h** (and several other headers)
- You can assign any pointer to NULL
- You can test a pointer to see if it's NULL
  - In fact, a NULL pointer evaluates to false, convenient.
- But, don't try to dereference it

```
int *ip = NULL;
double *dp = NULL;

if ( ip )
  …;

double bill = *dp;
```

Ouch

What are They Good For?

- To use some parts of the standard library, you have to use pointers.
  - Files, sorting, …
- We've already used them for pass-by-reference.
  - Now we can understand this technique
  - And use it for ourselves

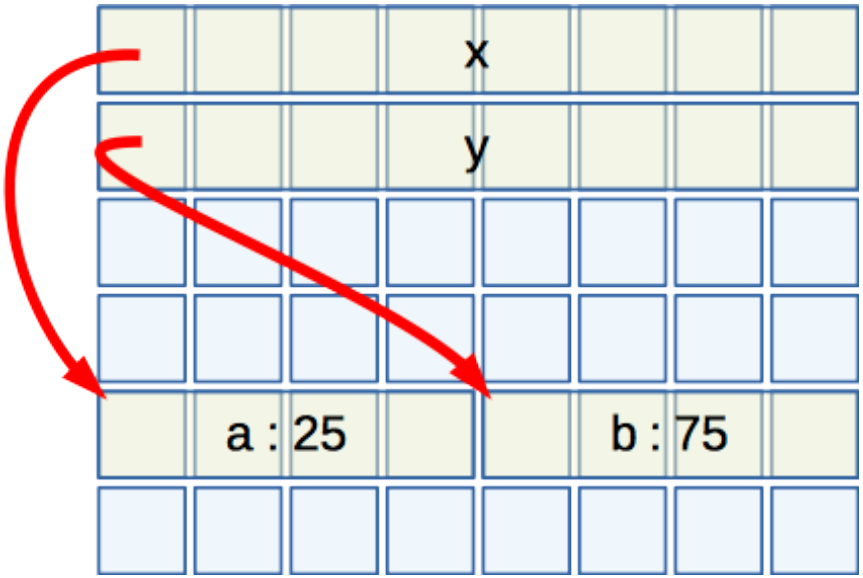
Pass-By-Reference

- By passing pointers to functions, we can let them modify the values pointed to.
- For example, we could write a working swap function.

```
void swap( int *x, int *y )
{
   int temp = *x;
   *x = *y;
   *y = temp;
}
```

You have to pass addresses to call it.

```
   swap( &a, &b );
```



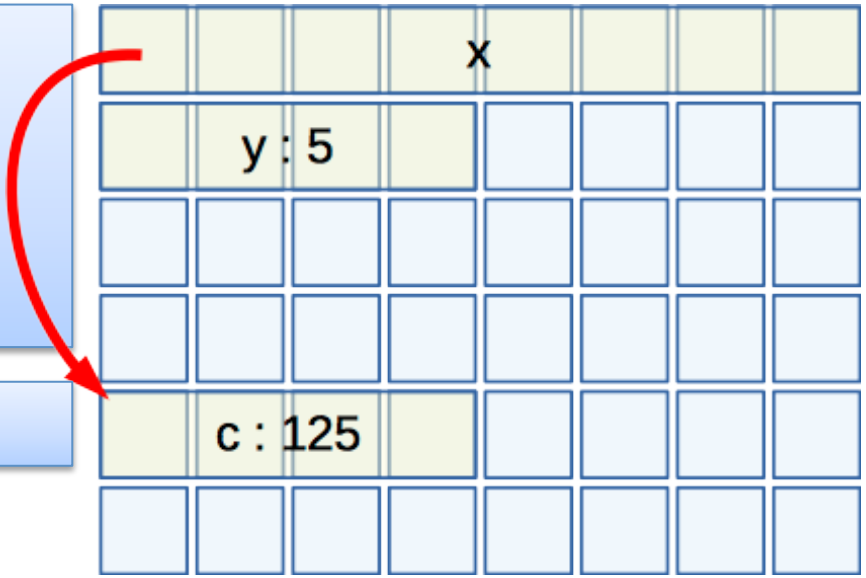
Pass-By-Reference

- Functions can take a mix of pointer an non-pointer parameter types.

Read this as, you need to pass me the address of an actual int.

```
void clamp(int *x, int bound)
{
  if ( *x > bound )
    *x = bound;
}
```

```
  clamp( &c, 5 );
```



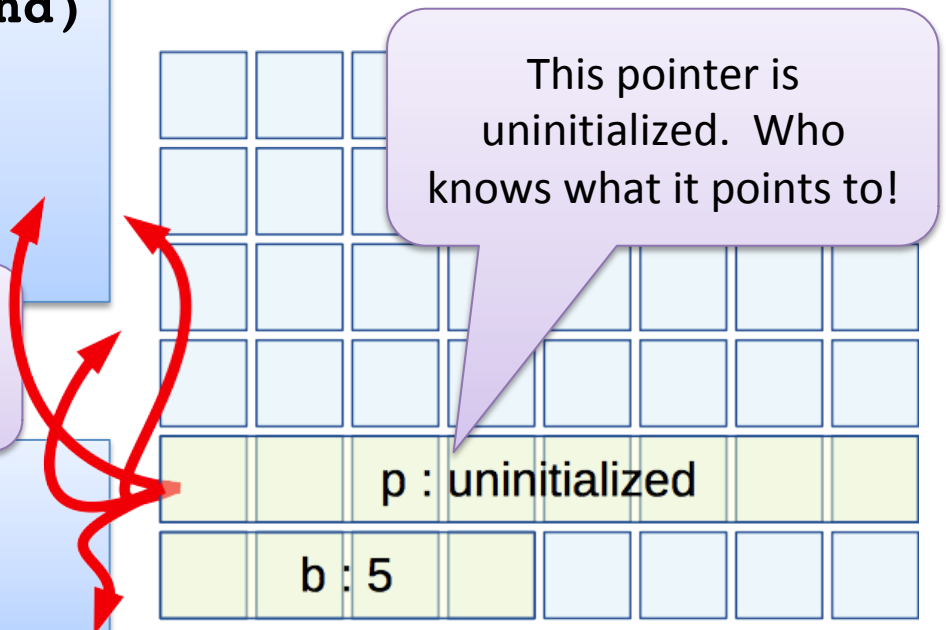
How To Segfault

- It's easy to make pointer mistakes, if you're just trying to get something that compiles

```
void clamp(int *x, int bound)
{
  if ( *x > bound )
    *x = bound;
}
```

```
  int *p;
  int b = 5;
  clamp( p, b );
```

"It says it wants a pointer to an int, so I guess I'll give it one."



How To Segfault

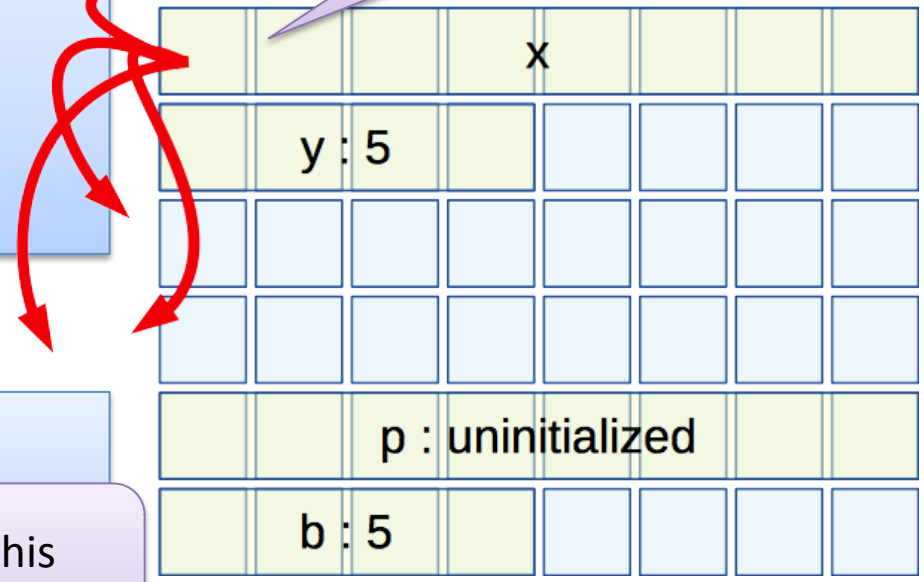
- This will compile … but bad things will happen when you run it.

```
void clamp(int *x, int bound)
{
  if ( *x > bound )
    *x = bound;
}
```

```
  int *p;
  int b = 5;
  clamp( p, b );
```

I've seen this mistake a lot ☹

Now. This pointer is a copy p (which was never initialized).

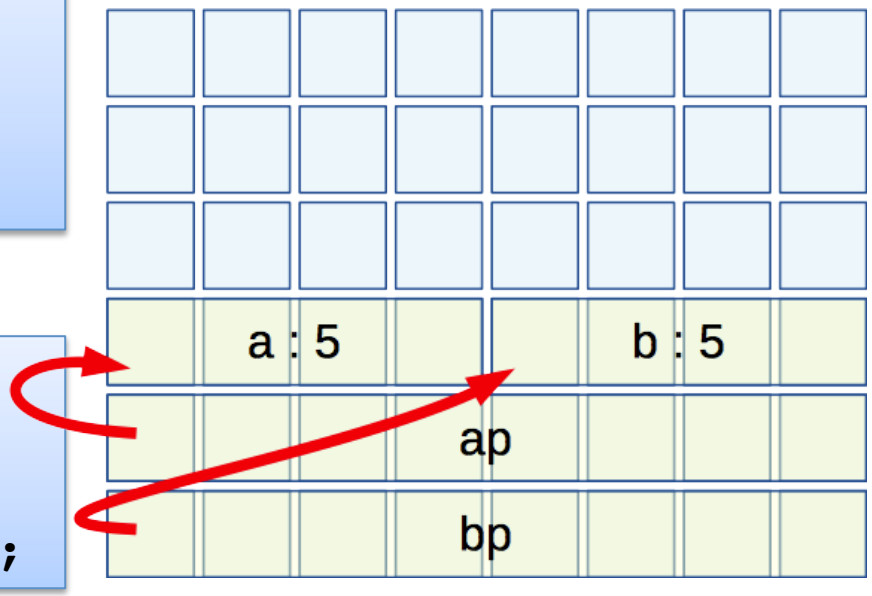


Still Pass-By-Value Underneath

- When you pass a pointer, you're still passing a value … just a value that happens to be an address.

```
void hulkSmash(int *x, int *y)
{
  x = NULL;
  y = NULL;
}
```

```
  int a = 5, b = 10;
  int *ap = &a, *bp = &b;
  hulkSmash( ap, bp );
  printf( "%d %d\n", *ap, *bp );
```



Still Pass-By-Value Underneath

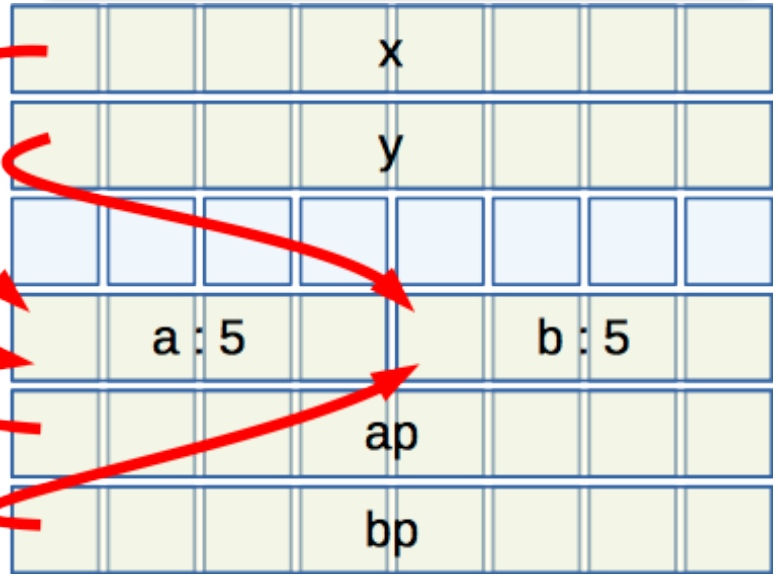
- When we call this function, it gets copies of pointers to a and b.

```
void hulkSmash(int *x, int *y)
{
  x = NULL;
  y = NULL;
}
```

This function is just throwing away its copies of these pointers.

But, the original pointers in main() are unchanged.

```
  int a = 5, b = 10;
  int *ap = &a, *bp = &b;
  hulkSmash( ap, bp );
  printf( "%d %d\n", *ap, *bp );
```



Returning a Pointer

- A function can return a pointer.
  - Giving the caller a chance to look at the value it points to.
  - Or even to change it.

```
int list[ 100 ];

int *find( int v )
{
  for (int i = 0; i < 100; i++)
    if ( list[ i ] == v )
      return &list[ i ];
  return NULL;
}
```

I don't just give you a copy of the value you're looking for.

… I tell you where it lives.

```
   int *p = find( 30 );
```

In this example, p could be null.

Realistic Pointer Values

- Really, something like 12 or 26 isn't a typical memory address.
  - On a 32-bit machine, a pointer has, well, 32 bits.
    - So a pointer might look like: 0xAC8B2E38
  - On a 64-bit machine, a pointer has 64 bits.
    - So a pointer might look like: 0xAC8B2E38F3274E38
- We can print out the numeric value of a pointer:

```
printf( "%p\n", ptr )
```

- Pointers generally all take the same amount of storage … but not all pointers are the same.

Type Matters … Sometimes

- Recall, C will let you convert values all day long without a complaint.

```
char c = 'a';
float f = 1.23;
short s = 123;

c = f;
f = s;
s = c;
```

No warnings here.

The compiler knows to convert among these types.

Type Matters … Sometimes

- But, for pointer types, it will give you a warning

```
char c = 'a';
float f = 1.23;
short s = 123;

char *cp = &c;
float *fp = &f;
short *sp = &s;

fp = &c;
sp = &f;
cp = &s;
```

These look good.

But here, are you sure you know what you're doing?

- A short pointer isn't the same as an float pointer.

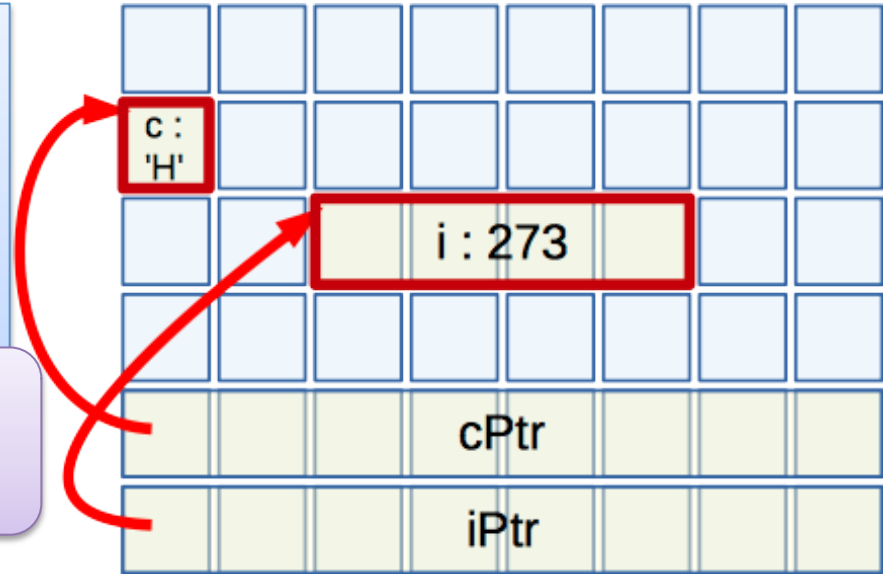
Pointers and Size

- The pointer type keeps up with the type of the thing pointed to.
  - Including the size.

```
char c = 'H';
int i = 273;

char *cPtr = &c;
int *iPtr = &i;

*cPtr = 'J';
*iPtr = 291;
```

I change 1 byte.

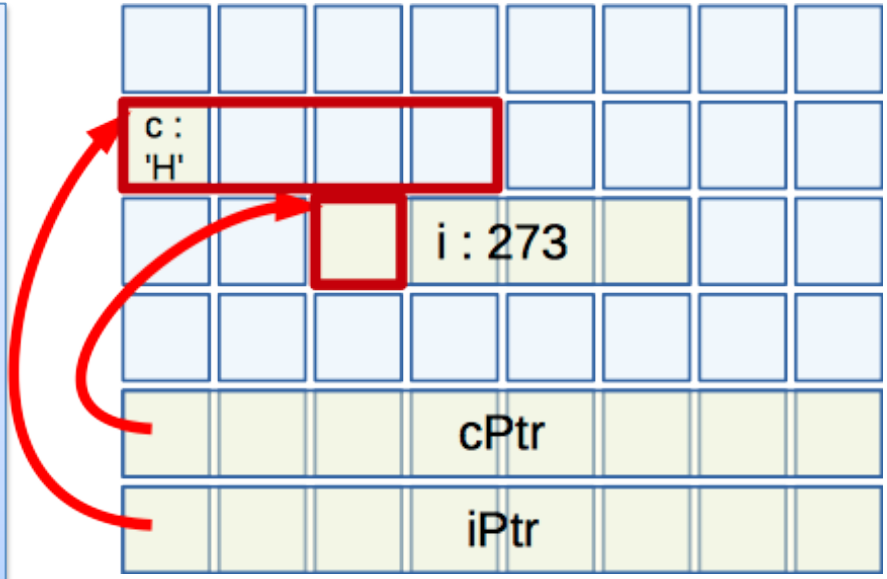
I change 4 bytes.



Pointers and Size

- With a cast, you can suppress compiler warnings for pointer type conversion.
  - But, you may write nonsense into memory.

```
char c = 'H';
int i = 273;

char *cPtr;
int *iPtr;

cPtr = (char *) &i;
iPtr = (int *) &c;

*cPtr = 'J';
*iPtr = 291;
```



Fun with Pointer Types

- In general, a pointer variable only wants a pointer to the same type of value.

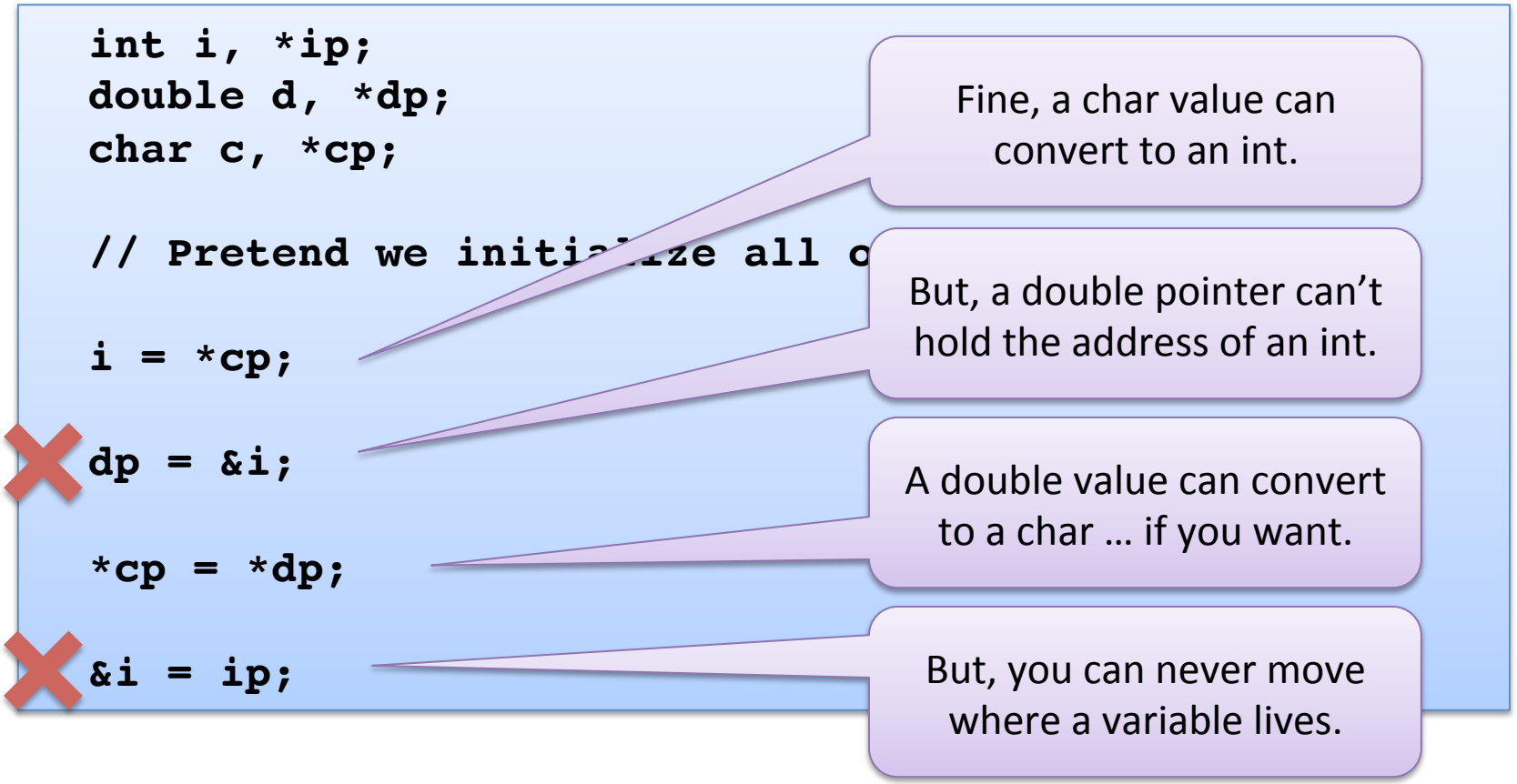
```
int i, *ip;
double d, *dp;
char c, *cp;

// Pretend we initialize all of these

i = *cp;

dp = &i;

*cp = *dp;

&i = ip;
```

You can declare pointers and values at the same time. The * goes with the name.

Which of these make sense?

Fun with Pointer Types

- In general, a pointer variable only wants a pointer to the same type of value.



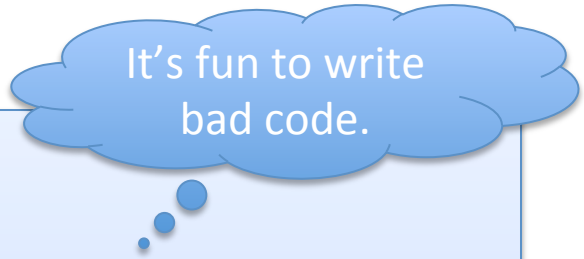
Sense …

- Some pointer operations make sense.

```
int a = 30, b = 50, *px = &a, *py = &b;

a = *py;   // Copy the value py points to into a

*px = 35;  // Copy 35 into the value pointed to by px

*px = b;   // Copy the value of b into the value px
           // points to.

*px = *py; // Copy the value py points to into the
           // value px points to.

px = &b;   // Copy the address of b into px

px = py;   // Make px point to the same thing as py
```

.. and Nonsensibility

- others, not so much.




```
int a = 30, b = 50, *px = &a, *py = &b;

px = &35;   // Numbers don't have addresses
a = *35;    // Constants don't make good pointers
px = 35;    // Constants still don't make good pointers
a = &b;     // An int isn't good for holding an address
a = *b;     // b (int) doesn't make a good pointer
*a = 35;    // An int still doesn't make a good pointer
&a = py;    // Can't move where a lives.
a = py;     // a (int) isn't good for holding an address
a = **py;   // *py (int) doesn't make a good pointer
px = b;     // b (int) doesn't make a good address
```

Pointers and Precedence

- Both * and & have very high precedence
- Almost the highest category

Operator	Description
++ -- () []	Postincrement, postdecrement : e.g., a++ Function call : e.g., f(5) Array index : e.g., a[5]
++ -- + - ! (*type*) * & sizeof	Preincrement, predecrement : e.g., --a Unary positive, negative : e.g., -a Logical not: e.g., !a Type cast: e.g., (float) a Pointer dereference: e.g., *p Address-of : e.g., &a Sizeof operator : e.g., sizeof(int)
* / %	Multiply, divide, mod

Inventing Types

- In C, * does three jobs
  - It's a binary infix operator, for doing multiplication
  - It's a unary prefix operator, for dereference pointers
  - It's a type operator, for defining new types
- Just like [ ] lets us define array types.
- In fact, we can invent infinitely many new types … if you gave me enough time.

```
int *p;
int a[] = { 1, 2 };
int **pp;
int *ap[ 10 ];
```

I'm a pointer to an int.

I'm a pointer to a pointer to an int.

I wonder what I am.

Reading Types

- We can create some complex types
  - Without even defining classes
  - And, it will get ~~worse~~ … more interesting.
- We need to be able to understand the types we're creating or looking at.
- Two simple rules for reading types
  - Start at the variable name and work your way out.
  - Respect type operator precedence (which is the same as operator precedence)

```
int * * pp;
```

```
int * ap [ 10 ];
```

Meet `const`

- In C, const is another type operator.
  - It says, “Don’t let me change this thing’s value during its lifetime.”

```
int a = 25;
int const b = a + 1;
const int c = b + 1;
```

It reads better with the const here.

But you can put it here. Most people do.

Const isn't Constant

- Const variables can get a new value every time they are initialized
- So, they aren't considered compile-time constants.

```
for ( int i = 0; i < 10; i++ ) {
   const int c = i + 10;
   ...
}
```

You can't use me as a case in a switch statement.

Const and Pointers

- With const, we can say what we'd like to be able to do via a pointer.
  - Remember how to read types.

I'm a pointer to a const integer.

I'm a const pointer to an integer.

I'm a const pointer to a const integer.

```
int a = 25, b = 35, c = 45;

int const * x = &a;

int * const y = &b;

int const * const z = &c;
```

Const and Pointers

- With const, we can say what we'd like to be able to do via a pointer.
  - Remember how to read types.

```
int a = 25, b = 35, c = 45;

int const * x = &a;
*x = 26;
int * const y = &b;
y = &c;
int const * const z = &c;
z = &b;
*z = 44;
```

You can't do this.

.. or this

.. or either of these

Why Use Const?

- Const is useful in function types. It says what the function plans to do with your values.

```
int f( double const *d );
```

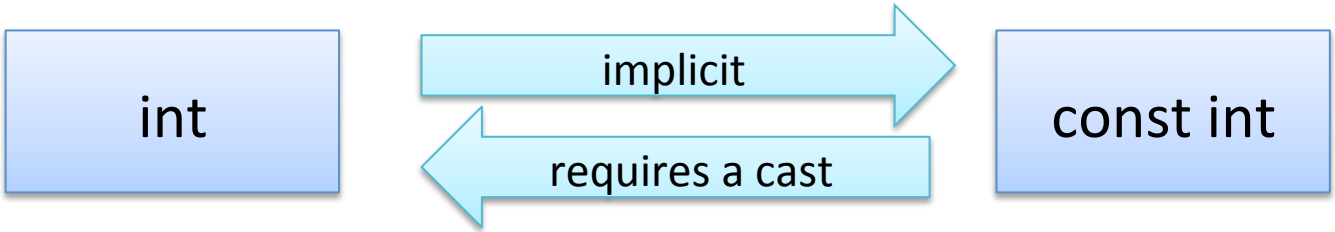
I'd like to use your double, but I promise not to change it.

See, I just want to look at what d points to.

```
int f( double const *d )
{
  int x = *d * *d * 3.14;
  return x;
}
```

Using const

- Const is just another part of the type (like ‘unsigned’)
- There are conversion rules for const-ness.



- We say a program is *const correct* when:
  - Functions use const to say which (reference) parameters they might modify
  - And whether or not their return (by reference) value can be modified

Trying to Cheat

- If you try to violate const-ness, the compiler will notice.

```
int f( double const *d )
{
   *d += 1;
    return 5;
}
```

This will be a compile error.

Cheating

- But, const is part of the type … and you can type cast it away.

```
int f( double const *d )
{
   * (double *) d += 1;
    return 5;
}
```

OK. I guess you know what you're doing.

- Think of const as a mechanism for programmers trying to cooperate …
  - … not a security measure.