Pointers Part 1

If we use a pointer and the value it points to no longer exists, **Undefined Behaviour** will occur

Memory is a huge array of byte-sized memory cells, each with an address

Each variable v is stored in memory in `sizeof(v)` adjacent memory cells

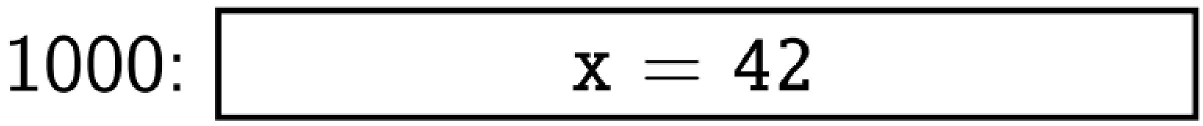
The address of a variable v is written `&v`

This is the address of the **first** of those memory cells that v occupies

There may also be alignment constraints on a particular architecture - that any 32-bit int (that occupies 4-bytes) has to have an address divisible by 4, for instance

Suppose we have a simple variable, located at address 1000:

`int x = 42;`



Then we add the following declaration and initialisation:
`int *ip = &x;`

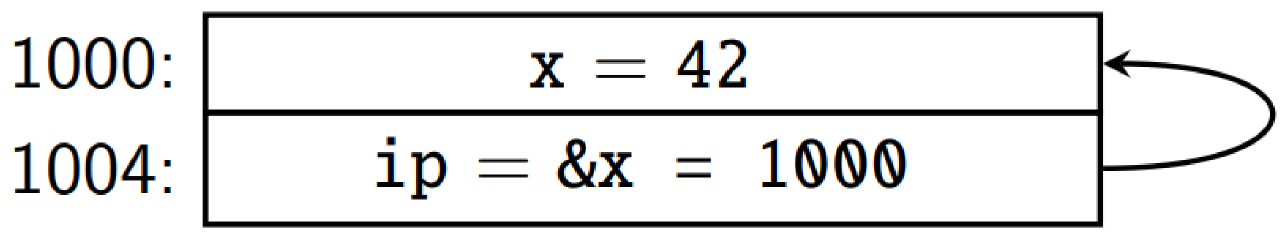
First, the declaration: `int *ip;` declares a variable called `ip` that is an `int *` - a pointer to an int

It’s value can be the address of any int variable

Second, the initialization: `ip = &x;` sets the value of `ip` to the address of x - to 1000

That is: it makes `ip` **point at** `x`

Assuming that x occupies 4 bytes, `ip` will be placed at address 1004. So now our memory layout is:

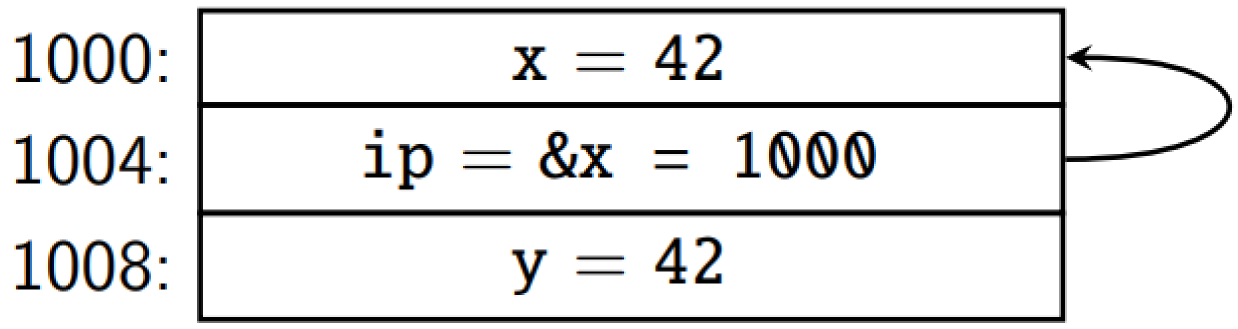


Next if we declare another int variable, and initialize it:

`int y = *ip;`

`*ip` is called the dereference operator: it means fetch the value of `ip` (1000) and then go read the value of the int stored at address 1000 (i.e. 42). Then we set y to the result: y = 42

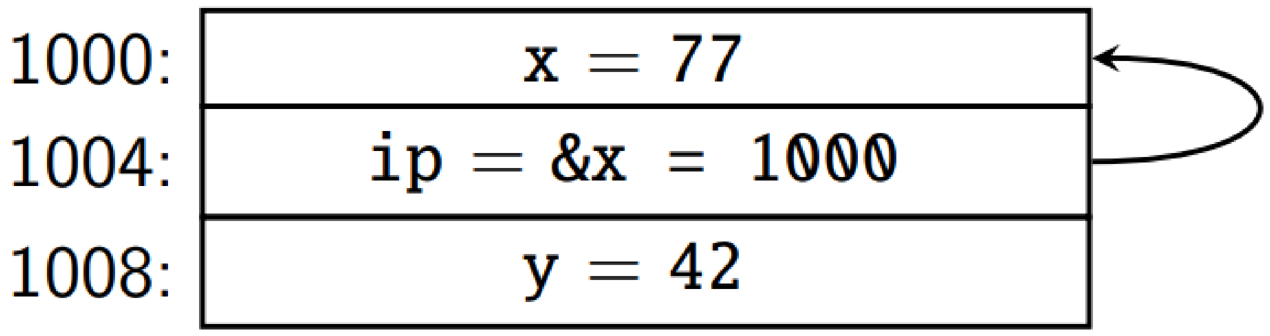
Our memory layout is now as follows (assuming pointers are 4 bytes too):



`ip` was `&x` so `*ip` must be `*&x`, and '`*`' and '`&`' are inverse operators so cancel out, hence `*ip == *&x == x`

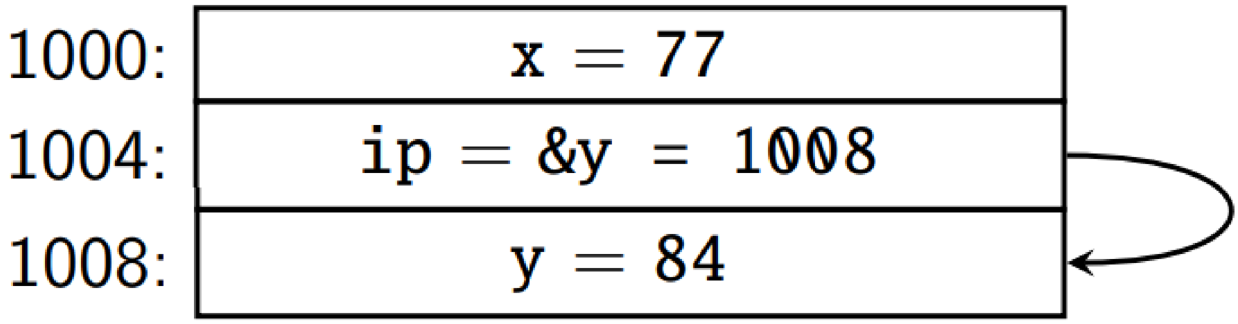
You can assign to `*ip` as well, which will alter x via the pointer:

`*ip = 77;`



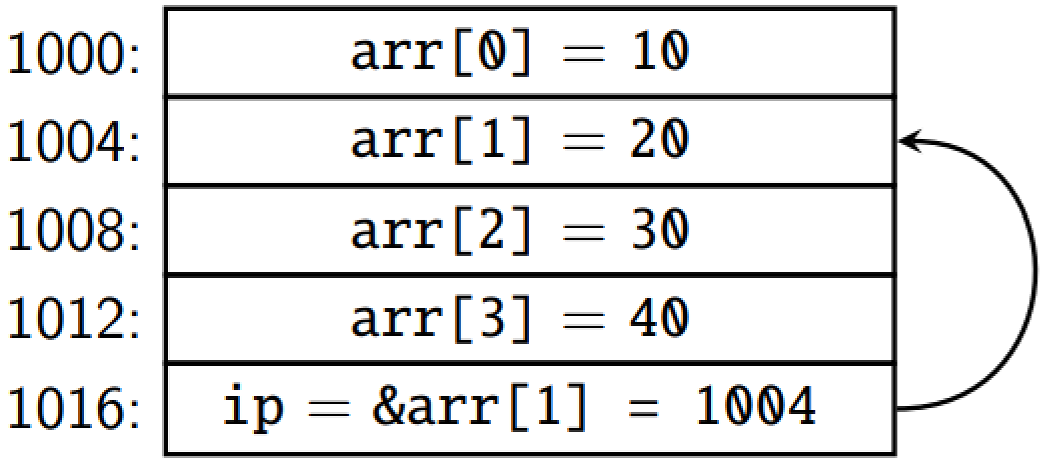
`ip` doesn't have to always point at `x`. If we add:

```
ip = &y;
*ip *= 2;
```



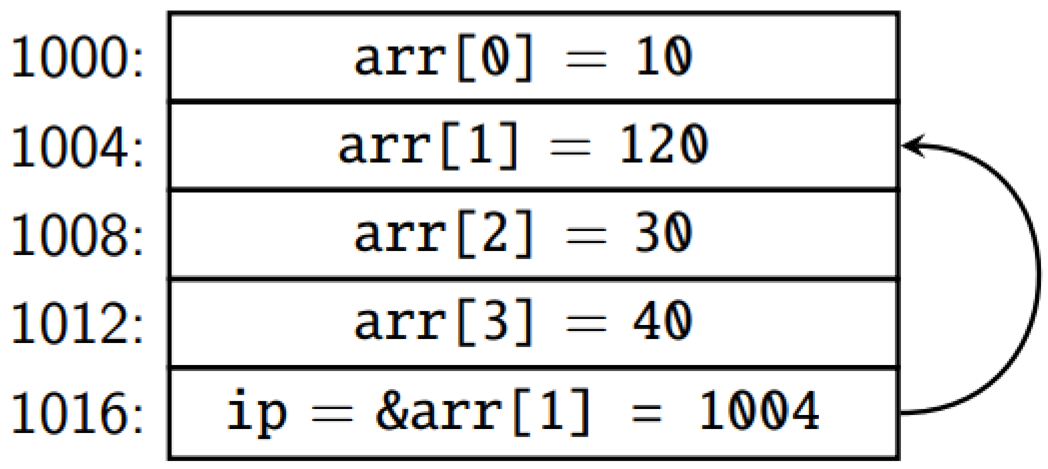
We could make `ip` point at an array element:

```
int arr[] = {10,20,30,40};
int *ip = &arr[1];
```



Then we can add 100 to the element via the pointer:

`*ip += 100;`



`++` is stronger than `*`

Only declare one pointer variable per line

Basal Decay

We've seen that we can make `ip` point at an array element. Suppose we make it point at the first array element:

```
int arr[] = {10,20,30,40};
int *ip = &arr[0];
```

`&arr[0]` can be written more simply as `arr`:
`int *ip = arr;`

This always happens in C: whenever we write an array name without a following `[`, the array name silently turns into a **basal pointer** - a pointer to the first array element. This process is called **basal decay**

`printf` can also print pointer values using the "%p" specifier, which prints it as an appropriate width hexadecimal integer. For instance:

```
int arr[] = {10,20,30,40};
ip = &arr[1];
printf( "arr=%p, ip=%p, *ip=%d\n", arr, ip, *ip );
```

Gives:

```
arr=0x7ffd393595a0, ip=0x7ffd393595a4, *ip=20
```

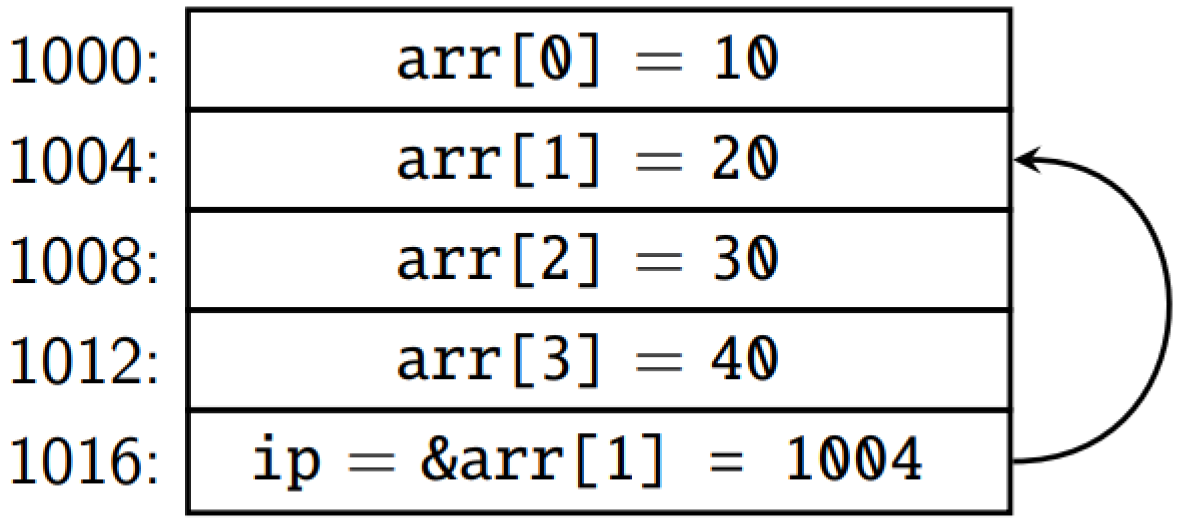
Each time we run this, the base address of the array changes slightly. This is because gcc arranges to start the stack at slightly different positions each time, in order to make memory addresses unpredictable - to make hackers' lives more difficult

Pointer Arithmetic

When a pointer (such as `ip`) points within an array, we can advance it to point at the next array element by writing `ip++`

So:

```
int arr[] = {10,20,30,40};
int *ip = arr;
ip++;
```



Note that `ip++` (and all pointer arithmetic) is scaled

It doesn't just add 1 to ip's numeric value, that would increment 1000 to 1001. But 1001 is not a valid address of any int, because it's not divisible by 4! It's certainly not `&arr[1]` which is what we want. Instead, `ip++` is scaled by `sizeof(*ip)`, here it adds `sizeof(int)==4` to `ip`

`ip` was 1000 (`&arr[0]`), and changes to 1004 (`&arr[1]`)

This gives us a simple, idiomatic, pointer-based `strlen(s)`:

```
int len = 0;
for( char *p = s; *p != ’\0’; p++ ) {
        len++;
}
```

Note that C makes no distinction between a pointer to one element of type T (a Pto1) and a pointer to one of several adjacent elements of type T - typically a pointer within an array (a PtoM)

In both cases, the type is `T *`. But we, the programmers, do need to make such a distinction

It is a classic error to do any pointer arithmetic when we’re dealing with a Pto1. Pointer arithmetic (`p++` etc) is only allowed when we’re dealing with a PtoM

If we’re unwise the same pointer variable may point to single elements for a while, then be made to point within an array later. Pointer arithmetic is then permissible during the second part - but not the first

Decide whether a particular pointer variable is a Pto1 or a PtoM

Even when we have a PtoM, it is our responsibility to make sure that we don’t fall off either end of the underlying array

We can increment a PtoM by N elements:
`ip += N;`

You can’t add two pointers together - that’s meaningless

We can also move a PtoM back one element, or by N elements:

```
ip--;
ip -= N;
```

All of these are scaled as well - so the value of `ip` has `sizeof(*ip)*N` added or subtracted from it. In all cases, it’s your responsibility to ensure that you don’t fall off the beginning or the end of the array within which `ip` points

If you do, it’s **Undefined Behaviour** - which means that your program is guaranteed not to be portable

You’re allowed to make a PtoM point to `&arr[N]` where the array has N elements

This is a pointer one element beyond the end of the array, it's allowed because many loops naturally lead to this final value. But you're **not allowed to dereference that pointer**

`&arr[N]` == `arr+N`, for any N

Symbolically, if we dereference both sides we get: `*&arr[N] == *(arr+N)` which simplifies to: `arr[N] == *(arr+N)`

Given:
`char strbuf[] = "hello everyone";`

The following three accesses are identical:

```
char letter1 = strbuf[7];
char letter2 = *(strbuf+7);
char letter3 = (strbuf+4)[3];
```

This is true whether or not strbuf is an array or a PtoM (`char *`):

```
char *s = strbuf;
char letter1 = s[7];
char letter2 = *(s+7);
char letter3 = (s+4)[3];
```

Pointer Comparisons

Two PtoMs, pointing within the same array, can be compared using the usual numeric comparison operators (`==` , `!=` , `<` etc)

Given:

```
int arr[] = {10,20,30,40,50,60};
int *ip = &arr[1]; // arr+1
ip++; // arr+2
ip += 2; // arr+4
```

The following comparisons will succeed or fail as shown:

```
if( ip == arr ).. // will fail
if( ip == &arr[4] ).. // will succeed
if( ip == arr+4 ).. // will succeed
if( ip > arr+2 ).. // will succeed
if( ip < arr+3 ).. // will fail
```

We can use pointer comparison and a one step beyond pointer to show a purely pointer-based implementation of summing up an array:

```
int arr[] = {10,20,30,40,50,60};
int *beyond = arr + NELEMENTS(arr); // one step beyond the end of
arr
int total = 0;
for( int *ip = arr; ip < beyond; ip++ ) // for each element in arr
{
        total += *ip;
}
```

Similarly, if we have two PtoMs, pointing within the same array, we can subtract the smaller from the larger to find out the number of elements between them:

```
int arr[] = {10,20,30,40,50,60};
int *ip = arr+3;
int n = ip-arr;
if( n == 3 ) // will succeed
```

For both comparison and subtraction, it is **Undefined Behaviour** if the pointers are not PtoMs that both point within the same array

`*p++` is called **post-increment**, it delivers the current value of `*p` and then increments the value of p as a side-effect

For Example:

```
char msg[] = "hi!";
char *p = msg;
char ch = *p++; // ch = 'h', increments p to msg+1 aka &msg[1]
char ch2 = *p++; // ch2 = 'i', increments p to msg+2 aka &msg[2]
```

This ends up with `ch = ’h’`, `ch2 = ’i’`, and `p = msg+2`

Can’t we just write `ch = *p; p++` instead? Yes we can, but `ch = *p++` is an expression that can be embedded in bigger expressions

This enables the most compact and idiomatic implementation of `strcpy(dst,src)`:
`while( (*dst++ = *src++) != ’\0’ ) /*EMPTY BODY*/;`

This is equivalent to:

```
char ch;
do {
        ch = *src; src++;
        *dst = ch; dst++;
} while( ch != ’\0’ );
```

`*p--`: post-decrement; this delivers `*p` and then decrements p
`*++p`: called pre-increment; this first increments p and then delivers `*p`
`*--p`: called pre-decrement this first decrements p and then delivers `*p`

Pointers, Strings and String Literals?

```
printf( "hello, x=%d\n", x );
strcpy( strbuf, "Hi!" );
char strbuf[] = "hello there";
```

But, now that we know that a `char *` (if it’s a PtoM) can be used to point to the first character in a string, we observe that it is also legal to write:
`char *str = "hello there";`
Similar to `char strbuf[] = "hello there";`, but what is the difference?

In `char strbuf[] = "hello there";`
The string literal is used to initialize the array
Exactly the right amount of storage is allocated on the stack (assuming this is a local variable) and initialized with successive characters from the string literal - not forgetting the ’`\0`’ terminator

In `char *str = "hello there";` all that is allocated on the stack is space for a pointer. The C compiler finds somewhere - usually in a **read-only** portion of memory - to store the string literal (plus it's '`\0`' terminator), and then arranges to initialize `str` with the address of the first character of the literal in memory

So `sizeof(strbuf)` is 12, whereas `sizeof(str)` is `sizeof(char *)`

The most important difference is that `strbuf` **may be safely modified**

Whereas `str` cannot be `strcpy()`ed into, because it points at a string literal: which is a **read-only string**

If you attempt to write into a read-only string literal, this leads to a segmentation fault or **Undefined Behaviour** again

Using string literals in `printf()` and `strcpy()` (second argument) was OK - those functions were not going to modify those strings

Summary of Pointer-to-One (Pto1)

Pto1 Declarations:

Description	**Example**	**Notes**
Declaration	`T *p`	declare p as a ptr-to-T
DeclareAndInit	`T *p = pto1value`	declare p as a ptr-to-T, and initialise it

Pto1 Values:

Description	**Example**	**Notes**
AddressOf	`&var`	address of *var* in memory
AddressOfEl	`&array[N]`	address of that array element in memory
PtrVar	`p1`	current value of *p1* (where *p1* is a Pto1 variable)

Pto1 Operations:

Description	**Example**	**Notes**
Assign	`p = pto1value`	copy *pto1value* to *p*
Dereference	`*p`	dereference *p*, deliver the value to which it points

Summary of Pointer-to-Many (PtoM)

Pto1 Declarations:

Description	Example	Notes
Declaration	`T *p`	declare p as a ptr-to-T
DeclareAndInit	`T *p = ptomvalue`	declare p as a ptr-to-T, and initialise it

PtoM Values:

Description	Example	Notes
PtrVar	`pm`	current value of *pm*
1DArray	`arrname`	basal pointer to 1st element of array
PtrPlusN	`pm+N`	ptr to N elements after where *pm* points
PtrMinusN	`pm-N`	ptr to N elements before where *pm* points
ArrayPlusN	`arrname+N`	ptr to Nth element in array
AddressOfEl	`&arrname[N]`	equivalent to previous entry
StrLit	`"hello"`	a `char *`, basal pointer to a readonly string

(where *pm* is a PtoM variable)

PtoM Operations:

Description	Example	Notes
Assign	`pm = ptomvalue`	copy *ptomvalue* to *pm*
Dereference	`*pm`	dereference *pm*, deliver value to which *pm* points
Increment	`pm++`	increment *pm* to point at next element
Decrement	`pm--`	decrement *pm* to point at previous element
PostIncrement	`*pm++`	Deliver `*pm`, then `pm++`
PostDecrement	`*pm--`	Deliver `*pm`, then `pm--`
PreIncrement	`*++pm`	First `pm++`, then deliver `*pm`
PreDecrement	`*--pm`	First `pm--`, then deliver `*pm`
PtrSubtract	`int gap = pm1-pm2`	Number of elements between *pm*1 and *pm*2
PtrComparison	`if( pm1 > pm2 )`	Comparison of pointer addresses
		you can use `'=='`, `'!='`, `'<'`, `'<='`, `'>'` and `'>='`

(where *pm* is a PtoM variable, *ptomvalue* is a PtoMvalue, and *pm*1 and *pm*2 are PtoMvalues that point into the same array)