Pointers

In the opposite direction to &, given a pointer value we can get at the value stored at that address using the * operator

```
int n = 1234, *pn = &n; // declaration with initialisation

printf("n has value %d, pn has value %p\n", n, pn);
printf("the value pn points to is %d\n", *pn);
*pn = 23;
printf("n has value %d, pn has value %p\n", n, pn);
```

Pointers

In the opposite direction to &, given a pointer value we can get at the value stored at that address using the * operator

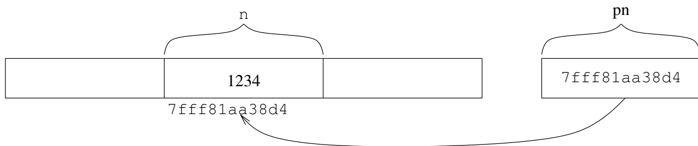
```
int n = 1234, *pn = &n; // declaration with initialisation

printf("n has value %d, pn has value %p\n", n, pn);
printf("the value pn points to is %d\n", *pn);
*pn = 23;
printf("n has value %d, pn has value %p\n", n, pn);
```

Produces

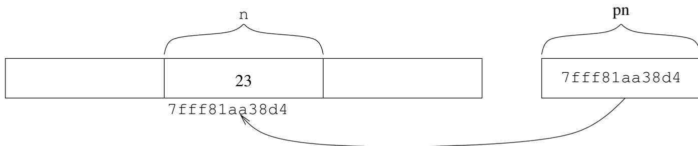
```
n has value 1234, pn has value 0x7fff81aa38d4
the value pn points to is 1234
n has value 23, pn has value 0x7fff81aa38d4
```

Pointers



Initial values

Pointers



After `*pn = 23`

Pointers

The ∗ operator says “the value in this variable is a pointer; operate on the value at that address”

Pointers

The ∗ operator says “the value in this variable is a pointer; operate on the value at that address”

Following a pointer to find the value at the address it indicates is called *indirecting* through that pointer

Pointers

The $*$ operator says "the value in this variable is a pointer; operate on the value at that address"

Following a pointer to find the value at the address it indicates is called *indirecting* through that pointer

The type of the pointer tell us how to interpret the bytes at that address

Pointers

The $*$ operator says “the value in this variable is a pointer; operate on the value at that address”

Following a pointer to find the value at the address it indicates is called *indirecting* through that pointer

The type of the pointer tell us how to interpret the bytes at that address

So if the variable has type pointer to integer, it will address 4 bytes of integer; if the variable has type pointer to double, it will address 8 bytes of double; and so on

Pointers

The $*$ operator says "the value in this variable is a pointer; operate on the value at that address"

Following a pointer to find the value at the address it indicates is called *indirecting* through that pointer

The type of the pointer tell us how to interpret the bytes at that address

So if the variable has type pointer to integer, it will address 4 bytes of integer; if the variable has type pointer to double, it will address 8 bytes of double; and so on

This is a big reason why pointers to different types are distinguished: to determine how many bytes of value to access

Pointers

It is important to realise that `*pn = 99` does not modify the value of variable `pn`, but the value at the address contained by `pn`

Pointers

It is important to realise that `*pn = 99` does not modify the value of variable `pn`, but the value at the address contained by `pn`

So `*pn = 99` means "get the value of `pn`; store 99 at that address"

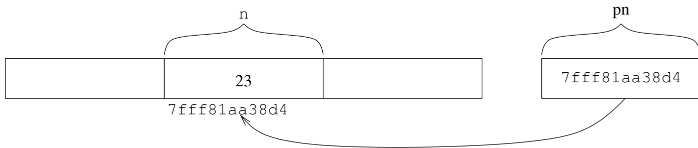
Pointers

It is important to realise that `*pn = 99` does not modify the value of variable `pn`, but the value at the address contained by `pn`

So `*pn = 99` means “get the value of `pn`; store 99 at that address”

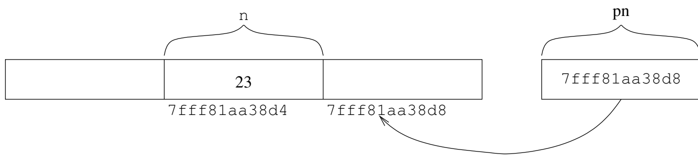
And modifying `pn` does not move `n` in memory; just that `pn` now points to somewhere else

Pointers



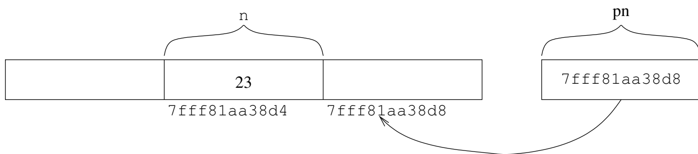
Initial values

Pointers



After `pn = 0x7fff81aa38d8`

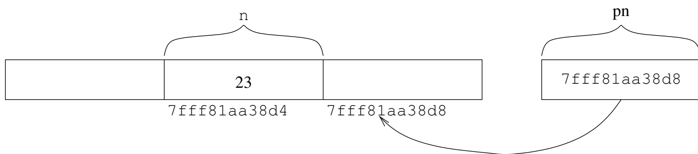
Pointers



After `pn = 0x7fff81aa38d8`

Now accessing `*pn` will access the four bytes at the address 0x7fff81aa38d8 (four bytes, as `pn` is an `int` pointer)

Pointers

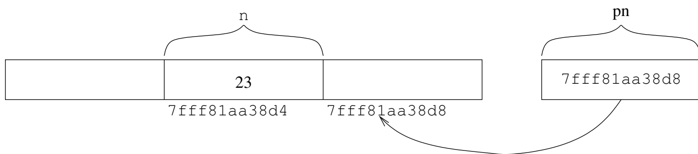


After `pn = 0x7fff81aa38d8`

Now accessing `*pn` will access the four bytes at the address 0x7fff81aa38d8 (four bytes, as `pn` is an `int` pointer)

Danger here if those four bytes are being used by something else in the program!

Pointers



After `pn = 0x7fff81aa38d8`

Now accessing `*pn` will access the four bytes at the address 0x7fff81aa38d8 (four bytes, as `pn` is an `int` pointer)

Danger here if those four bytes are being used by something else in the program!

C does not stop you messing with *any* bytes in memory!

Pointers

Note that these assignments never change the *locations* of variables, merely their *values*

Pointers

Note that these assignments never change the *locations* of variables, merely their *values*

If I changed the address I have for you in my address book, that doesn't make you move house!

Pointers

Note that these assignments never change the *locations* of variables, merely their *values*

If I changed the address I have for you in my address book, that doesn't make you move house!

I would just have the address of something that isn't your house

Pointers

Note that these assignments never change the *locations* of variables, merely their *values*

If I changed the address I have for you in my address book, that doesn't make you move house!

I would just have the address of something that isn't your house

Pointers can point anywhere in memory, they are not restricted to point at locations of variables

Pointers

Note that these assignments never change the *locations* of variables, merely their *values*

If I changed the address I have for you in my address book, that doesn't make you move house!

I would just have the address of something that isn't your house

Pointers can point anywhere in memory, they are not restricted to point at locations of variables

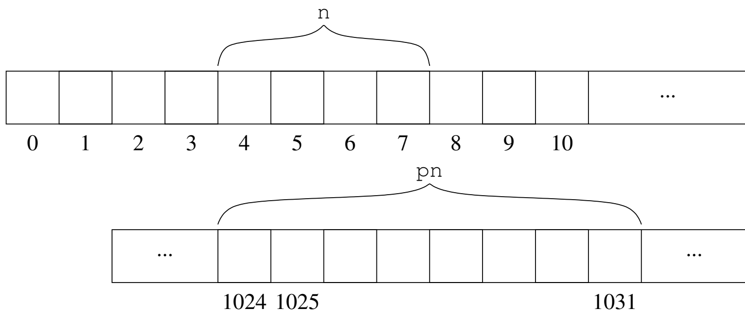
In fact, most useful applications of pointers are *not* pointing at variables

Pointers

As `pn` is a perfectly normal variable, it will be associated with some memory location

Pointers

As `pn` is a perfectly normal variable, it will be associated with some memory location



8 bytes of address on this 64-bit machine

Pointers

Note: different machines and operating systems may have different size of pointers, according to the needs of the particular machine

Pointers

Note: different machines and operating systems may have different size of pointers, according to the needs of the particular machine

General-purpose 32-bit machines will likely have pointers of size 4 bytes

Pointers

Note: different machines and operating systems may have different size of pointers, according to the needs of the particular machine

General-purpose 32-bit machines will likely have pointers of size 4 bytes

A major gotcha when porting poorly-written programs from 32 to 64 bit architectures

Pointers

Note: different machines and operating systems may have different size of pointers, according to the needs of the particular machine

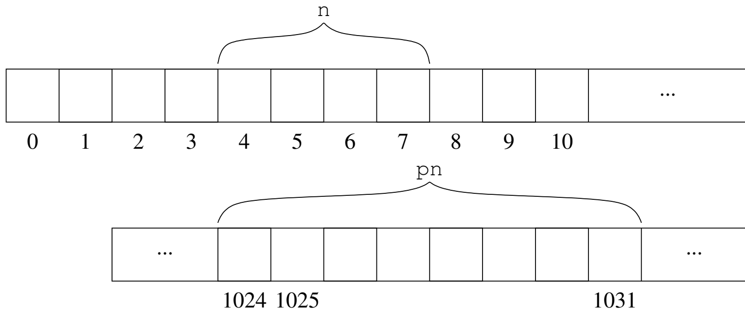
General-purpose 32-bit machines will likely have pointers of size 4 bytes

A major gotcha when porting poorly-written programs from 32 to 64 bit architectures

Some versions of C on early computers had 2 byte (16 bit) pointers: they didn't have enough memory to make 4 byte pointers necessary!

Pointers

On a 64-bit architecture:



Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer "4", namely the address of `n`

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer “4”, namely the address of `n`

But we need not stop here: `pn` is a variable, so it has an address: `&pn` is 1024 in this picture

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer “4”, namely the address of `n`

But we need not stop here: `pn` is a variable, so it has an address: `&pn` is 1024 in this picture

Then `&pn` is a pointer to a pointer to an integer

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer "4", namely the address of `n`

But we need not stop here: `pn` is a variable, so it has an address: `&pn` is 1024 in this picture

Then `&pn` is a pointer to a pointer to an integer

```
int **ppn = &pn;
```

declares `ppn` to be of type pointer to pointer to integer, and initialises it with a value, namely the address of the variable `pn`

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer "4", namely the address of `n`

But we need not stop here: `pn` is a variable, so it has an address: `&pn` is 1024 in this picture

Then `&pn` is a pointer to a pointer to an integer

```
int **ppn = &pn;
```

declares `ppn` to be of type pointer to pointer to integer, and initialises it with a value, namely the address of the variable `pn`

Use multiple `*`s as appropriate to the number of "pointer to"s

Pointers

What does this assignment do?

```
**ppn = 100;
```

Pointers

What does this assignment do?

```
**ppn = 100;
```

This will update the value of an integer at address 4 (namely `n`): `*ppn` retrieves the address of `pn`, then `**ppn` follows that pointer to the address of an integer, the value 100 is then stored there

Pointers

What does this assignment do?
`**ppn = 100;`

This will update the value of an integer at address 4 (namely `n`): `*ppn` retrieves the address of `pn`, then `**ppn` follows that pointer to the address of an integer, the value 100 is then stored there

Updating `*ppn` will change the address stored in `pn`, e.g., `*ppn = 8;` is the same as `pn = 8;`

Pointers

What does this assignment do?
`**ppn = 100;`

This will update the value of an integer at address 4 (namely `n`): `*ppn` retrieves the address of `pn`, then `**ppn` follows that pointer to the address of an integer, the value 100 is then stored there

Updating `*ppn` will change the address stored in `pn`, e.g., `*ppn = 8;` is the same as `pn = 8;`

Now `pn` no longer points at `n` but another place in memory

Pointers

Exercise (harder). The following does not work. Explain why and fix it using pointers.

```
void swap(int a, int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
...
int n = 1, m = 2;
swap(n, m);
printf("n = %d m = %d\n", n, m);
```

Pointers

One of Java's design principles was to eliminate pointers. In reality, it *does* have pointers, it just hides the fact from the naive programmer

Pointers

One of Java's design principles was to eliminate pointers. In reality, it *does* have pointers, it just hides the fact from the naive programmer

And makes it harder for experienced programmers

Pointers

One of Java's design principles was to eliminate pointers. In reality, it *does* have pointers, it just hides the fact from the naive programmer

And makes it harder for experienced programmers

Exercise. Find out how Java manages pointers

Exercise. Is it possible to write a (primitive) integer swap function in Java?

Arrays and Pointers

Pointers are intimately associated with arrays in C

Arrays and Pointers

Pointers are intimately associated with arrays in C

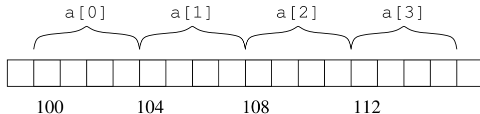
Consider an array `int a[4];`

Arrays and Pointers

Pointers are intimately associated with arrays in C

Consider an array `int a[4];`

In memory, C arrays are laid out simply



To be definite, we fix on using 4 byte (32 bit) integers

Arrays and Pointers

Adjacent members of the array are adjacent in memory

Arrays and Pointers

Adjacent members of the array are adjacent in memory

If the array starts at address 100, so `a[0]` is at address 100, then `a[1]` is at address $100 + \texttt{sizeof(int)} = 104$; `a[2]` is at address $100 + 2 \times \texttt{sizeof(int)} = 108$; and so on

Arrays and Pointers

Adjacent members of the array are adjacent in memory

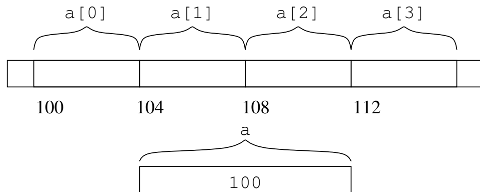
If the array starts at address 100, so `a[0]` is at address 100, then `a[1]` is at address $100 + \texttt{sizeof(int)} = 104$; `a[2]` is at address $100 + 2 \times \texttt{sizeof(int)} = 108$; and so on

Array element n is at address

$$100 + n \times \texttt{sizeof(int)}$$

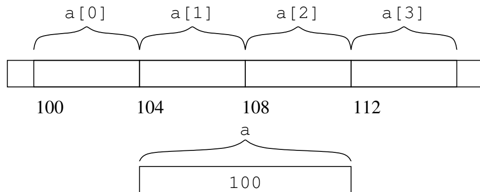
Arrays and Pointers

In fact the variable `a` contains the address of the start of the array



Arrays and Pointers

In fact the variable `a` contains the address of the start of the array



So `a` actually has type `int*` (with a caveat)

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

`a[3]` is the same as `*(a + 3)`

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

`a[3]` is the same as `*(a + 3)`

An important point here: **arithmetic on pointers is different from arithmetic on integers**

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

`a[3]` is the same as `*(a + 3)`

An important point here: **arithmetic on pointers is different from arithmetic on integers**

For an integer pointer `a` the expression `a + 3` is not simply the address 3 along from the value in `a`, but instead is the address of the *3rd integer* along

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

`a[3]` is the same as `*(a + 3)`

An important point here: **arithmetic on pointers is different from arithmetic on integers**

For an integer pointer `a` the expression `a + 3` is not simply the address 3 along from the value in `a`, but instead is the address of the *3rd integer* along

If the value in `a` is 100, and integers are of size 4, then `a + 3` is the address $100 + 3 \times 4 = 112$

Arrays and Pointers

This is strange at first, but turns out to be what you always want

Arrays and Pointers

This is strange at first, but turns out to be what you always want

This is another reason to distinguish types of pointers. For a variable `v` of type `T*`, the expression `v + ` n is computed as

$$\texttt{v} + n \times \texttt{sizeof(T)}$$

Giving the address of the nth item along

Arrays and Pointers

This is strange at first, but turns out to be what you always want

This is another reason to distinguish types of pointers. For a variable `v` of type `T*`, the expression `v + ` n is computed as

$$\texttt{v} + n \times \texttt{sizeof(T)}$$

Giving the address of the nth item along

Pointer arithmetic counts in items, not bytes

Arrays and Pointers

The result is that for an array `a[]`

- the value of `a` is the address of the start of the array
- `*a` is the same as `a[0]`
- `a + 3` is the address of the item 3 further along (the 4th item)
- so `*(a + 3)` is the value there; just like `a[3]`
- `&(a[3])` is the address of that item; as is `a + 3`

Arrays and Pointers

The result is that for an array `a[]`

- the value of `a` is the address of the start of the array
- `*a` is the same as `a[0]`
- `a + 3` is the address of the item 3 further along (the 4th item)
- so `*(a + 3)` is the value there; just like `a[3]`
- `&(a[3])` is the address of that item; as is `a + 3`

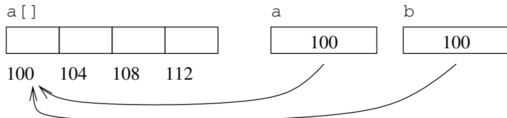
And exactly the same is true for a pointer variable, though this may or may not be pointing at the memory for an array

Arrays and Pointers

Also ++ works well. If we define an integer pointer

```
int *b;
b = a;
```

then `*b` (equivalently, `b[0]`) is the same as `a[0]`



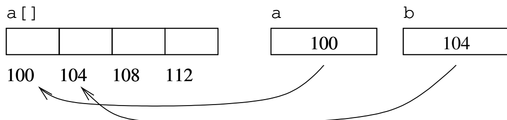
Arrays and Pointers

Also ++ works well. If we define an integer pointer

```
int *b;
b = a;
```

then `*b` (equivalently, `b[0]`) is the same as `a[0]`

After we increment `b++;` which is the same as `b = b+1;` the pointer moves one **integer** along. Now `*b` is the same as `a[1]`



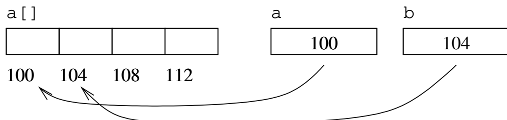
Arrays and Pointers

Also ++ works well. If we define an integer pointer

```
int *b;
b = a;
```

then `*b` (equivalently, `b[0]`) is the same as `a[0]`

After we increment `b++;` which is the same as `b = b+1;` the pointer moves one **integer** along. Now `*b` is the same as `a[1]`



So we can iterate `b` along the array using ++

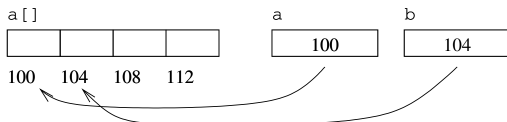
Arrays and Pointers

Also ++ works well. If we define an integer pointer

```
int *b;
b = a;
```

then `*b` (equivalently, `b[0]`) is the same as `a[0]`

After we increment `b++;` which is the same as `b = b+1;` the pointer moves one **integer** along. Now `*b` is the same as `a[1]`



So we can iterate `b` along the array using ++

The increment operator is an array iteration operator (as well as the normal increase-by-one on usual integers)

Arrays and Pointers

```
for (b = a; ...  ; b++) {
  ...  *b ...
}
```

is a common sight in C programs

Arrays and Pointers

```
for (b = a; ...  ; b++) {
  ...  *b ...
}
```

is a common sight in C programs

Conversely, for a pointer `b` we can write also `b[i]` just as if `b` was the name of an array

Arrays and Pointers

```
for (b = a; ...  ; b++) {
  ...  *b ...
}
```

is a common sight in C programs

Conversely, for a pointer `b` we can write also `b[i]` just as if `b` was the name of an array

The `x[n]` syntax really is just a short way of writing `*(x + n)` and is equally applicable to both arrays and pointers

Arrays and Pointers

```
for (b = a; ...  ; b++) {
  ...  *b ...
}
```

is a common sight in C programs

Conversely, for a pointer `b` we can write also `b[i]` just as if `b` was the name of an array

The `x[n]` syntax really is just a short way of writing `*(x + n)` and is equally applicable to both arrays and pointers

expr1[expr2] is the same as *(expr1 + expr2)

Arrays and Pointers

Exercise.

```
int n = 7;
int *p = &n;
```

Is `p[0]` meaningful?

Arrays and Pointers

In a very real sense, C does not have arrays as a basic type!

Arrays and Pointers

In a very real sense, C does not have arrays as a basic type!

Just integers, floating point, and pointers (which are integers)

Arrays and Pointers

In a very real sense, C does not have arrays as a basic type!

Just integers, floating point, and pointers (which are integers)

And `[]` and `""` are just some convenient syntax

Arrays and Pointers

In a very real sense, C does not have arrays as a basic type!

Just integers, floating point, and pointers (which are integers)

And `[]` and `""` are just some convenient syntax

Not necessarily a useful point of view, though!

Arrays and Pointers

Exercise. Explain

```
void copy(int *dst, int *src, int len)
{
  while (len--) {
    *dst++ = *src++;
  }
}
...
int a[128], b[128];
...
copy(b, a, 128);
```

Arrays and Pointers

Exercise. Then think through

```
copy(a, a + 64, 64);
```

Arrays and Pointers

We can now see the close identification of pointers and arrays

Arrays and Pointers

We can now see the close identification of pointers and arrays

In the declaration `int a[4];` we can regard the variable `a` to have type "array of integer", or as type "pointer to integer"

Arrays and Pointers

We can now see the close identification of pointers and arrays

In the declaration `int a[4];` we can regard the variable `a` to have type "array of integer", or as type "pointer to integer"

In all but a few useful circumstances the two concepts are interchangeable

Arrays and Pointers

We can now see the close identification of pointers and arrays

In the declaration `int a[4];` we can regard the variable `a` to have type "array of integer", or as type "pointer to integer"

In all but a few useful circumstances the two concepts are interchangeable

We can use pointers as arrays and arrays as pointers

Arrays and Pointers

We can now see the close identification of pointers and arrays

In the declaration `int a[4];` we can regard the variable `a` to have type “array of integer”, or as type “pointer to integer”

In all but a few useful circumstances the two concepts are interchangeable

We can use pointers as arrays and arrays as pointers

As long as you understand what you are doing

Except for a few subtleties

Array types and pointer types are interchangeable

Arrays and Pointers

Except for a few subtleties

Array types and pointer types are interchangeable

Array types are a special subset of pointer types: they are pointers that point at pre-allocated blocks of memory

Arrays and Pointers

This identification makes it easy to see what happens when we mis-index an array

Arrays and Pointers

This identification makes it easy to see what happens when we mis-index an array

Given the example above (`int a[4];`), what is `a[10]`?

Arrays and Pointers

This identification makes it easy to see what happens when we mis-index an array

Given the example above (`int a[4];`), what is `a[10]`?

It is `*(a + 10)`, namely the value stored at address $100 + 10 \times 4 = 140$, regarded as an integer

Arrays and Pointers

This identification makes it easy to see what happens when we mis-index an array

Given the example above (`int a[4];`), what is `a[10]`?

It is `*(a + 10)`, namely the value stored at address $100 + 10 \times 4 = 140$, regarded as an integer

This is beyond the end of the memory reserved by the system for the array `a`

Arrays and Pointers

This might

Arrays and Pointers

This might

- successfully return some integer value from whatever happens to be at that memory location

Arrays and Pointers

This might

- successfully return some integer value from whatever happens to be at that memory location
- this might be previously unused and uninitialised memory, or it might be where some other value (not necessarily an integer) is currently placed

Arrays and Pointers

This might

- successfully return some integer value from whatever happens to be at that memory location
- this might be previously unused and uninitialised memory, or it might be where some other value (not necessarily an integer) is currently placed
- or it might refer to an unmapped memory location (think about virtual memory and unmapped pages), when the OS might cause an interrupt and likely terminate your program

Arrays and Pointers

If your program crashes with a *segmentation violation* error or a *general protection fault*, this is likely what is happening: your program is reading or writing to an unexpected area of memory

Arrays and Pointers

If your program crashes with a *segmentation violation* error or a *general protection fault*, this is likely what is happening: your program is reading or writing to an unexpected area of memory

Occasionally you will see *bus error* for the same kind of thing

Arrays and Pointers

If your program crashes with a *segmentation violation* error or a *general protection fault*, this is likely what is happening: your program is reading or writing to an unexpected area of memory

Occasionally you will see *bus error* for the same kind of thing

If this happens you need to look carefully at your program to find the error

Arrays and Pointers

If your program crashes with a *segmentation violation* error or a *general protection fault*, this is likely what is happening: your program is reading or writing to an unexpected area of memory

Occasionally you will see *bus error* for the same kind of thing

If this happens you need to look carefully at your program to find the error

Exercise for geeks. Read up on value *alignment*

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Compile your program using the `-g` option to include debugging information: `cc -Wall -g ...`

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Compile your program using the `-g` option to include debugging information: `cc -Wall -g ...`

`valgrind ./myprog`
will run the program `./myprog` checking *all* the program's accesses to memory

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Compile your program using the `-g` option to include debugging information: `cc -Wall -g ...`

`valgrind ./myprog`
will run the program `./myprog` checking *all* the program's accesses to memory

This will slow execution horribly, of course, but it will highlight when you do something stupid

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Compile your program using the `-g` option to include debugging information: `cc -Wall -g ...`

`valgrind ./myprog`
will run the program `./myprog` checking *all* the program's accesses to memory

This will slow execution horribly, of course, but it will highlight when you do something stupid

Thus `valgrind` is a way of inserting that slow checking that other languages do all the time

Arrays and Pointers

Clang users can use `-fsanitize=address` when compiling to insert code that does something similar

Arrays and Pointers

As mentioned, usually C does not check for these kids of errors as

Arrays and Pointers

As mentioned, usually C does not check for these kids of errors as

- the significant overhead of checking memory accesses

Arrays and Pointers

As mentioned, usually C does not check for these kids of errors as

- the significant overhead of checking memory accesses
- sometimes the programmer *does* want to write code that accesses off the nominal ends of an array; you can sometimes find code like `a[-1]`. This is valid C, and the programmer will get everything they deserve

Arrays and Pointers

In the declarations of an array and a pointer

```
int a[4];
int *b;
```

we need to be clear about what is happening

Arrays and Pointers

In the declarations of an array and a pointer

```
int a[4];
int *b;
```

we need to be clear about what is happening

`a` is a variable of type pointer to integer **and** a chunk of memory (e.g., 16 bytes) is reserved somewhere for the array; the value of the variable `a` will be the address of that chunk of memory

Arrays and Pointers

In the declarations of an array and a pointer

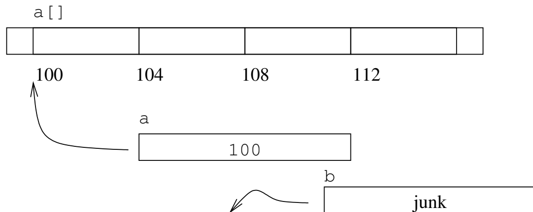
```
int a[4];
int *b;
```

we need to be clear about what is happening

`a` is a variable of type pointer to integer **and** a chunk of memory (e.g., 16 bytes) is reserved somewhere for the array; the value of the variable `a` will be the address of that chunk of memory

`b` is a variable of type pointer to integer, with no particular value, and no chunk of memory is reserved

Arrays and Pointers



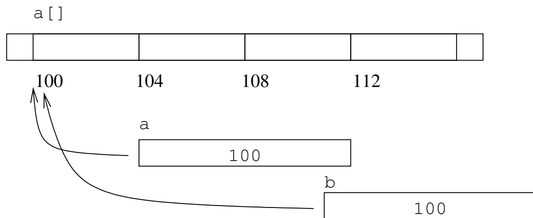
`int a[4]` sets up both `a` and the space for the array;
`int *b` just sets up `b`

Arrays and Pointers

`b` is a pointer variable, so we can set its value: `b = a;`

Arrays and Pointers

`b` is a pointer variable, so we can set its value: `b = a;`



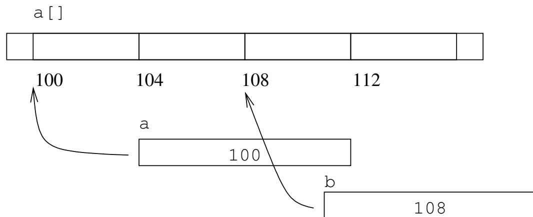
And now `b[1]` makes sense; it is the same as `a[1]`
`b[1]` is at address $100 + 1 \times 4 = 104$

Arrays and Pointers

We could equally do `b = a + 2`

Arrays and Pointers

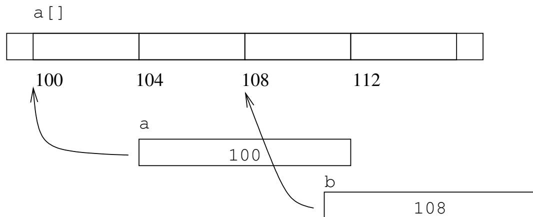
We could equally do `b = a + 2`



And now `b[1]` makes sense; it is the same as `a[3]`
`b[1]` is at address $108 + 1 \times 4 = 112$

Arrays and Pointers

We could equally do `b = a + 2`



And now `b[1]` makes sense; it is the same as `a[3]`
`b[1]` is at address $108 + 1 \times 4 = 112$

And now `b[-1]` makes sense; it is the same as `a[1]`
`b[-1]` is at address $108 + (-1) \times 4 = 104$