• computer memory is a large array of bytes
• a variable will be stored in 1 or more bytes
• on CSE machines a char occupies 1 byte, an int 4 bytes, a double 8 bytes
• The & (address-of) operator returns a reference to a variable.
• Almost all C implementations implement pointer values using a variable’s address in memory
• Hence for almost all C implementations & (address-of) operator returns a memory address.
• It is convenient to print memory addresses in Hexadecimal notation.
int k;
int m;

printf("address of k is %p\n", &k);
// prints address of k is 0xbfffffb80

printf("address of m is %p\n", &m);
// prints address of k is 0xbfffffb84

• k occupies the four bytes from 0xbfffffb80 to 0xbfffffb83
• m occupies the four bytes from 0xbfffffb84 to 0xbfffffb87
Arrays in Memory

Elements of the array will be stored in *consecutive* memory locations:

```c
int a[5];

int i = 0;
while (i < 5) {
    printf("address of a[%d] is %p\n", i, &a[i]);
}
// prints:
// address of a[1] is 0x7ffe693d61c4
// address of a[2] is 0x7ffe693d61c8
// address of a[3] is 0x7ffe693d61cc
// address of a[4] is 0x7ffe693d61d0
```
Size of a Pointer

Just like any other variable of a certain type, a variable that is a pointer also occupies space in memory. The number of bytes depends on the computer’s architecture.

- 32-bit platform: pointers likely to be 4 bytes  
  e.g. older operating systems/machines
- 64-bit platform: pointers likely to be 8 bytes  
  e.g. CSE machines, many student machines
- tiny embedded CPU: pointers could be 2 bytes  
  e.g. your microwave
Pointers

A pointer is a data type whose value is a reference to another variable.

```c
int *ip;    // pointer to int
char *cp;   // pointer to char
double *fp; // pointer to double
```

In most C implementations, pointers store the memory address of the variable they refer to.
• The \& (address-of) operator returns a reference to a variable.
• The * (dereference) operator accesses the variable referred to by the pointer.

For example:

```c
int i = 7;
int *ip = &i;
printf("%d\n", *ip); // prints 7
*ip = *ip * 6;
printf("%d\n", i); // prints 42
i = 24;
printf("%d\n", *ip); // prints 24
```
Pointers

- Like other variables, pointers need to be initialised before they are used.
- Like other variables, it's best if novice programmers initialise pointers as soon as they are declared.
- The value NULL can be assigned to a pointer to indicate it does not refer to anything.
- NULL is a `#define` in `<stdio.h>`
- NULL and 0 interchangeable (where a pointer is expected).
- Most programmers prefer NULL for readability.
We’ve seen that when primitive types are passed as arguments to functions, they are passed by value and any changes made to them are not reflected in the caller.

```c
void increment(int n) {
    n = n + 1;
}
```

This attempt fails. But how does a function like `scanf` manage to update variables found in the caller? `scanf` takes pointers to those variables as arguments!

```c
void increment(int *n) {
    *n = *n + 1;
}
```
We use pointers to pass variables *by reference*! By passing the address rather than the value of a variable we can then change the value and have the change reflected in the caller.

```c
int i = 1;
increment(&i);
printf("%d\n", i);
//prints 2
```

In a sense, pointer arguments allow a function to ‘return’ more than one value. This greatly increases the versatility of functions. Take `scanf` for example, it is able to read multiple values and it uses its return value as error status.
Classic Example
Write a function that swaps the values of its two integer arguments.

Before we knew about pointer arguments this would have been impossible, but now it is straightforward.

```c
void swap(int *n, int *m) {
    int tmp;
    tmp = *n;
    *n = *m;
    *m = tmp;
}
```
You should not find it surprising that functions can return pointers. However, you have to be extremely careful when returning pointers. Returning a pointer to a local variable is illegal - that variable is destroyed when the function returns. But you can return a pointer that was given as an argument:

```c
int increment(int *n) {
    *n = *n + 1;
    return n;
}
```

Nested calling is now possible: `increment(increment(&i));`
A C array has a very simple underlying representation, it is stored in a contiguous (unbroken) memory block and a pointer is kept to the beginning of the block.

```c
char s[] = "Hi!";
printf("s: %p *s: %c\n\n", s, *s);
printf("&s[0]: %p s[0]: %c\n", &s[0], s[0]);
printf("&s[1]: %p s[1]: %c\n", &s[1], s[1]);
printf("&s[2]: %p s[2]: %c\n", &s[2], s[2]);
printf("&s[3]: %p s[3]: %c\n", &s[3], s[3]);

// prints
// s: 0x7fff4b741060 *s: H
// &s[0]: 0x7fff4b741060 s[0]: H
// &s[1]: 0x7fff4b741061 s[1]: i
// &s[2]: 0x7fff4b741062 s[2]: !
// &s[3]: 0x7fff4b741063 s[3]:
```

Array variables act like pointers to the beginning of the array!
Because array variables act like pointers, when we passed them to functions we can change the array. We can also use pointers like array names if they point at an array:

```c
int nums[] = {1, 2, 3, 4, 5};
int *p = nums;

printf("%d\n", nums[2]);
printf("%d\n", p[2]);
// both print: 3
```
We can even make a pointer point to the middle of an array:

```c
int nums[] = {1, 2, 3, 4, 5};
int *p = &nums[2];
printf("%d %d\n", *p, p[0]);
```

There are differences between an array variable and a pointer.

```c
int i = 5;
p = &i; // this is OK
nums = &i; // this is an error
```

Unlike a pointer, an array variable is constant and may not be modified.
It always points to the start of the array. Of the array, it
Arrays are converted to pointers when pass as function parameters

// all 3 prototypes are equivalent
void print_array(int length, int array[length]);
void print_array(int length, int array[]);
void print_array(int length, int *array);

The first prototype is more readable but the length is ignored in the 2nd parameter.
Pointers can be tested for equality or relative order.

```c
double ff[] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
double *fp1 = ff;
double *fp2 = &ff[0];
double *fp3 = &ff[4];

printf("%d %d\n", (fp1 > fp3), (fp1 == fp2));
// prints: 0 1
```

Note that we are comparing the values of the pointers, i.e., memory addresses, not the values the pointers are pointing to!
Pointer Summary

Pointers:

- are a compound type
- usually implemented with memory addresses
- are manipulated using address-of(&) and dereference(*)
- should be initialised when declared
- can be initialised to NULL
- should not be dereferenced if invalid
- are used to pass arguments by reference
- are used to represent arrays
- should not be returned from functions if they point to local variables
The C View of Data

Variables are examples of *computational objects*.
Each computational object has

- a *location* in memory (obtainable via `&`)
- a *value* (ultimately just a bit-string)
- a *name* (unless created by `malloc()`)
- a *type*, which determines ...
  - its *size* (in units of whole bytes, `sizeof`)
  - how to *interpret* its value; what *operations* apply
- a *scope* (where it’s visible within the program)
- a *lifetime* (during which part of program execution it exists)
The C View of Data

Memory regions during Linux gcc/clang program execution ...

machine code for program goes here
global vars and constants go here
malloc’d objects go here
local vars, parameters go here
The C View of Data

Example of runtime stack during call to h()

```c
int main() {
    int n, m;
    n = 5;  m = f(n);
}
int f(int x) {
    return g(x);
}
int g(int y) {
    int r = 4 * h(y);
    return r;
}
int h(int z) {
    int i, p = 1;
    for (i=1; i<=z; i++)
        p = p * i;
    return p
}
```
Exercise: Properties of Variables

Identify the properties of each of the named objects:

```c
int a;    // global int variable

int main(void) {
    int b;          // local int variable
    char c;         // local char variable
    char d[10];     // local char array
    ...
}

int e;    // global int variable

int f(int g) {          // function + parameter
    double h;          // local double variable
    ...
}
```
The Physical View of Data

Memory = indexed array of bytes

Each byte contains 8 bits

Memory is a very large array of bytes

Indexes are “memory addresses” (a.k.a. pointers)
Properties of physical memory

- called main memory (or RAM, or primary storage, ...)
- indexes are “memory addresses” (a.k.a. pointers)
- data can be fetched in chunks of 1, 2, 4, 8 bytes
- cost of fetching any byte is same (ns)
- can be volatile or non-volatile

When addressing objects in memory ...

- any byte address can be used to fetch 1-byte object
- byte address for $N$-byte object must be divisible by $N$
Memories can be categorised as *big-endian* or *little-endian*.

![Diagram showing big-endian and little-endian memory layouts.](image)
Arrays

Arrays are defined to have $N$ elements, each of type $T$

Examples:

```c
int a[100]; // array of 10 ints
char str[256]; // array of 256 chars
double vec[100]; // array of 100 doubles
```

Elements are laid out adjacent in memory

```
int a[4];
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

&=1000 &=1004 &=1008 &=100C
Arrays

Assuming an array declaration like `Type v[N]` ...

- individual array elements are accessed via indices 0..N−1
- total amount of space allocated to array $N \times \text{sizeof}(Type)$
- array name gives address of first element (e.g. $v = &v[0]$)
- array indexing can be treated as $v[i] \approx *(v+i)$

Strings are just arrays of `char` with a ‘\0’ terminator

- constant strings have ‘\0’ added automatically
- string buffers must allow for element to hold ‘\0’
Arrays

When arrays are “passed” to a function, actually pass &a[0]

```c
int main(void)
{
    char str[5] = “abc”;
    f(str);
}

void f(char *s)
{
    while (*s != ‘\0’)
        printf(“%c”, *s++);
}
```
Arrays can be created automatically or via malloc()
**Structs**

Structs are defined to have a number of components:
- each component has a *Name* and a *Type*

Example:

```
typedef struct ... Date;

struct _student {
    int id;
    char given[50];
    char family[50];
    char gender;
    Date bday;
};
```

*Defines a new data type called* `struct _student`
Internal layout of struct components determined by compiler

Each name maps to a byte offset within the struct
E.g. in first example id = offset 0, given = offset 4, family = offset 54, etc.
Structs

To ensure *alignment*, internal “padding” may be needed.

Padding wastes space; re-order fields to minimise waste.