Pointers
Pointers

Pointers are an address in memory
   Includes variable addresses, constant addresses, function address...

It is a data type just like any other (int, float, double, char)

On 32-bit machines, pointers are 4 bytes in size
   On 64-bit machines, pointers are 8 bytes

Pointers point to a particular data type
   The compiler checks pointers for correct use just as it checks int, float, etc.
Declaring pointers

No data type called pointer

Instead, use * to denote a pointer

```c
int *ptrx; // pointer to data type int
float *ft; // pointer to data type float
short *st; // pointer to data type short
```

Compiler associates pointers with corresponding data types

Variables `ptrx` and `ft` contain addresses that hold int and float values

How big (in bytes) are `ptrx`, `ft`, and `st`?
Referencing/dereferencing pointers

How can you create a pointer to a variable?

Use & which returns the address of the argument

```c
int y = 7;
int *x = &y; // assigns the address of y to x
```

How can you get the value pointed to?

Dereference the pointer using * (unf. * is used both in definitions and here)

Go to the address which is stored in x, and return the value at that address.

```c
int y = 7;  // y is 7
int *x = &y; // x is the memory address of y
int z = *x; // z is now 7
```

```c
(*a)++;  // increments the value pointed to a
*(a + 1);  // accesses the value pointed to by the address (a + 1)
```
Pointer quiz

```c
int y = 10;
int x = y;
y++;
x++;

What is the value of y?

int y = 10;
int *x = &y;
y++;
(*x)++;

What is the value of y?
```
Arrays and pointers

Compiler associates the address of the array to/with the name

```
int temp[34];
```

Array name (temp) is the pointer to the first element of array

To access the nth element of the array:

- Address = starting address + n * size of element
- Starting address = name of the array
- Size of element = size of data type of array
- `<array name>[n]` de-references the value at the nth location in the array

```
int temp[10]; // Assume temp = 100 (memory address)
temp[5] = *(100 + (4 x 5)) = *(120) // dereference address 120
```
Passing arrays

Passing an array passes a pointer

Passing an array as an argument passes the address

Hence arrays are always passed by reference

```c
int general (int size, int name []); //Expects a pointer to an int array
int general (int size, int *name);    //Expects a pointer to an int
```

```c
void foo(int a[]) {
    a[0] = 17;
}
```

```c
int b[1] = { 5 };
foo(b);
```

What is the value of `b[0]`?
Functions and pointers

Functions must return a value of the declared type

Just like variables, functions can return a pointer

What does the following function return?

```c
float *calc_area (float radius);
```

Function formal arguments may be of type pointer:

```c
double calc_size (int *stars);
```

For example, scanf takes in parameters as pointers:

```c
int scanf(const char *format, ...); // int*, int*
scanf("%d%f", &x, &f);
```
Passing in pointers

Why pass a variable address at all and complicate functions?
By design we can return only one value
Sometimes we need to return back more than one value

For example, consider `scanf("%d%f", &x, &f);`
Three values are returned (in `x`, `f`, and the return value)

Pointers allows us to return more than one value
Pointer arithmetic

Pointers can be added to and also subtracted from
Pointers contain addresses

Adding to a pointer goes to next specified location (dep. on data type)

\[
<data\ type> \ *\text{ptr};
\]

\[
\text{ptr + d means ptr + d * sizeof (<data type>)};
\]

For example

\[
\text{int} \ *\text{ptr};
\]
\[
\text{ptr + 2 means ptr + 2*4 which is ptr + 8}
\]

\[
\text{char} \ *\text{ptr};
\]
\[
\text{ptr + d means ptr + 2*1 which is ptr + 2}
\]
Example

```c
#include <stdio.h>

int main () {
    int *i;
    int j = 10;
    i = &j;
    printf("address of j is : %p\n", i);
    printf("address of j + 1 is : %p\n", i + 1);
}
```

What is the output?

```
$ ./a.out
address of j is : 0xbffffffa60
address of j + 1 is : 0xbffffffa64
$
```

Note that j + 1 is actually 4 more than j
Strings
Character strings

A sequence of character constants such as “This is a string”
  Each character is a character constant in a consecutive memory block

Representation in memory

```
This is a string \0
```

Each character is stored in ASCII, in turn is stored in binary
  Character strings are actually character arrays

A string constant is a character array whose elements cannot change
```
char *msg = "This is a message";
```
Strings as arrays

char *msg = "This is a string !";

The variable msg is associated with a pointer to the first element

msg is an array of 19 characters

\0 is also considered a character
  Appended to each string by the OS
  Used to distinguish strings in memory, acts as the end of the string
  Also called the NULL character

Character pointers

char *ptr;
ptr = "This is a string";

ptr is a character pointer containing the address of the first character (T)
  Which is the first element of the character array containing "This is a string"
String functions

Pointers to character strings can be manipulated as other pointers

```c
char *point1, *point2 = "welcome";
point1 = point2;
if (point1 == point2) { // valid, but will only compare pointers
```

Utilities provided as part of the C standard libraries

Most of the functions can be found in the header file string.h or stdlib.h

Always check the man page to find out the header file of that function

```
bash$ man 3 strlen
```
strcmp, strlen

### strcmp

```c
int strcmp (char *ptr1, char *ptr2)
```

*Compares strings pointed to by ptr1 and ptr2.*

*Returns 0 if identical strings, non-zero otherwise.*

```c
if (strcmp ("welcome", "cs132") == 0) { ... }
```

```c
char *ptr = "welcome";
if (strcmp ("welcome", ptr) == 0) // true
```

### strlen

```c
int strlen (const char *ptr)
```

*Returns count of characters in string.*

*Does not include NULL character in count.*

```c
int x = strlen ("welcome"); // x has value 7
```
strcpy

char *strcpy (char *ptr1, char *ptr2)

Copies entire string pointed to by ptr2 onto ptr1.
Returns address of string at ptr1 (we had this anyway, but we get it back anyway, useful sometimes)

char *ptr1 = “welcome”;
char ptr2 [10];
strcpy (ptr2, ptr1);

Now ptr2 has *a copy* of the string “welcome”

IMPORTANT: ptr1 must have enough space to contain the entire string

strcpy (ptr, “hello”); // RUN-TIME ERROR
Getting numbers from strings

```c
int atoi (const char *ptr);

Converts an alphanumerical string to an integer if possible
Returns 0 and sets global variable `errno` if an error occurs
```

```c
double atof (const char *ptr);

Converts an alphanumerical string to a double if possible
```

```c
int a = atoi("17"); //a is now 17
double b = atof("89.29393"); //b is now 89.29393
```
Structures
Structures

An aggregate data type which contains a fixed number of components

Declaration:

```c
struct name {
    // components
    // more components
};
```

For example

```c
struct dob {
    int month;
    int day;
    int year;
};
```

Each dob has a `month`, `day`, and `year` (ints) inside it
Using structures

Declare variables using `struct` keyword

All internal variables are allocated space

```c
struct dob d1, d2;
```

Access member values using ‘.’ notation

```c
d1.day = 10;
d2.year = 1976;
printf("%d\n", d2.year);
```

A structure can be assigned directly to another structure

```c
struct dob d1, d2;
d1.day = 10;
d1.month = 12;
d1.year = 1976;
d2 = d1; // now d2 has the same values as d1 for its fields.
```
Operations on structures

Cannot check to see if two structures are alike directly

```
struct dob d1, d2;
if (d1 == d2) // WRONG !!!
```

To compare, we need to check every internal value

Cannot print structures directly

Must print one field at a time

Pointers to structures use the ‘->’ notation

```
struct dob *d1;
d1->year = 1976;
d1->day = 26;
d1->month = 6;
```
A little more on structures

Can be initialized field by field or during declaration

```c
struct dob d1 = {26, 06, 1976};
```

Can create arrays of structures

```c
struct dob d1[10]; // array of 10 structs dob
```

And access them in the usual manner

```c
d1[1].day = 26;
d1[1].month = 6;
d1[1].year = 1976;
```
Making a structure into a type

Type definition allows an alias for an existing type identifier

```c
typedef type name;
```

For example

```c
typedef struct dob_s {
    int day;
    int month;
    int year;
} dob;
```

Now, can simply do

```c
dob my_dob;
dob.year = 17;
```
C command line
**argc and argv**

How can we access the command line?

Done using two variables `argc` and `argv`, passed as an argument to `main`

```c
int main (int argc, char *argv[])
```

- `argc` contains the total number of arguments, which includes the command
- `argv` contains the list of pointers to all the arguments (length `argc`)

Who fills up these two variables?

Done by the OS
- `argv` is automatically resized to include the whole command
Using the arguments

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    printf ("The number of arguments = %d\n", argc);
    for (i = 0; i < argc; i++)
        printf ("%d. %s\n", i, argv[i]); // print each argument.
}
```

Will print out each of the arguments passed in

```
bash$ ./a.out
    The number of arguments = 1
  0. ./a.out
bash$$ ./a.out first second
    The number of arguments = 3
  0. ./a.out
  1. first
  2. second
```