General Structure

stdio = standard input & output

```
#include <stdio.h>
#include <stdlib.h>
```

library imports

curly braces like java

```
int main(){

    //this is a comment
    // int main() is the main method in C

    return 0;
}
```

statements end w/semicolons

* print() or System.out.print() is now printf()
* newline is still \n

Comments

```
//single-line comment

/* this is a
   multi-line
   comment
*/
```

Format Specification(s)

ex. `printf("I am %d years old", 20);`

prints out: I am 20 years old

ex. `printf("%d rabbits ate %d carrots", 13, 42);`

prints out: 13 rabbits ate 42 carrots

%d is for DIGITS, and will floor floats/decimals (integers)

%f is for Floating point numbers

Can be used like so:

ex 1. `printf("grade: %f", 96.24);`

prints: grade: 96.240000

ex 2. `printf("grade: %0.2f", 96.24);`

prints: grade: 96.24

ex 3. `printf("grade: %0.1f", 96.24);`

prints: grade: 96.2

Format Specifiers

%c	character
%d	digit (signed integer)
%e or %E or %g or %G	scientific notation for floats
%f	float
%hi	signed integer (short)
%hu	unsigned integer (short)
%i	unsigned integer
%l or %ld or %li	long
%lf	double
%Lf	long double
%lu	unsigned int or unsigned long
%lli or %lld	long long
%llu	unsigned long long
%o	octal representation
%p	pointer
%s	string
%u	unsigned int
%x or %X	hexadecimal representation
%n	prints nothing
%%	prints % character

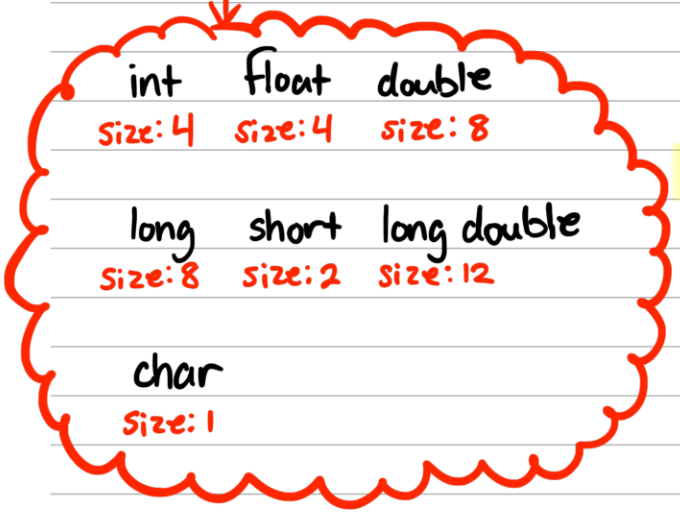
Variable Declaration

<type> <name>;

Declaration structure

ex. int age;

ex. age = 30; ← assignment after declaration



C Data Types

Reading User Input

- uses scanf

ex.

```
int grade1;
int grade2;
```

} variable declarations

```
scanf("%d", &grade1);
scanf("%d", &grade2);
```

} reads two integer inputs from console and stores in designated variables

Casting

ex.

```
int num = 2;

double Float_num = (double) num;
```

typecast

Conditionals

ex.

```
int grade = 20;

if (grade > 30)
    printf("y");

else
    printf("n");
```

alt. ex.

```
int grade = 20;

if (grade > 30) {
    printf("y");
}
else {
    printf("n");
}
```

Same functionality

*note: only works WITHOUT curly braces if action is singular.

*like Java, else if exists. ie.

```
if (condition)
else if (condition)
else
```

relational (comparison) operators are the same as usual:

greater? `>`	greater or equal? `>=`
less than? `<`	less than or equal? `<=`
`==` is equal?	`!=` not equal?

logical operators same as java:

`&&` AND	`||` OR

Assignment operators largely intuitive:

`=`	`+=`	`-=`
`i = 2`	`i+=2` so `i=4`	`i-=1` so `i=3`
`*=`	`/=`	`%=`
`i *= 4` so `i = 12`	`i/=2` so `i=6`	`i%=5` so `i=1`

Switch Statements

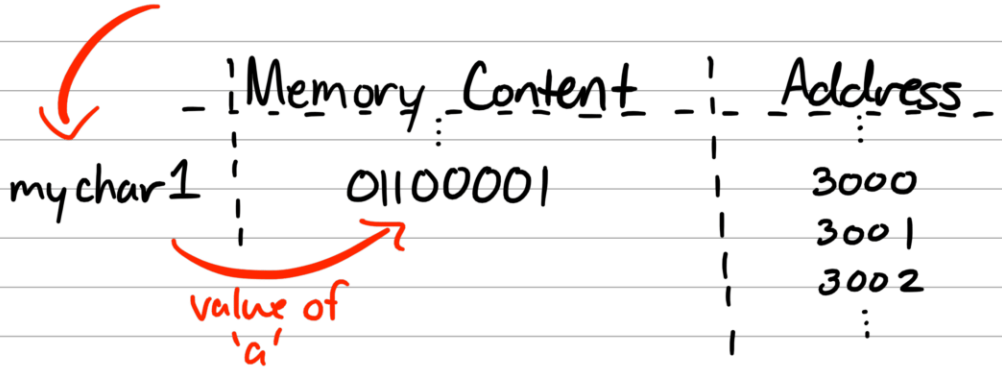
ex.

```
switch (grade) {

    case 'A':
        ~~~~;
        break;
    case 'B':
        ~~~~;
        break;
    case 'C':
        ~~~~;
        break;
    ~~etc.~~
        ~~~~;
        ~~~;

}
```

Memory

- related to ASCII and binary values of data

in the event of: mychar1 = 'a';



Loops

ex.

```
int count = 0;
while (count < 3) {
    printf("%d", count);
    count += 1;
}
```

condition

While Loop Example

ex.

```
int count = 2;
do {
    printf("%d", count);
} while (count < 2);
```

condition

do-while Loop Example
*runs code THEN checks condition

ex.

```
for(int i=0; i<5; i++) {
    printf("%d", i);
}
```

condition

for-loop Example

Functions

General Structure

<return type> <function name> <parameters>

signature structure

Return Types

- int
- float
- double
- short
- long
- char
- void

ex. simple

```
void myFunc() {
    printf("Hi!\n");
}
```

ex. parameters

```
void myFunc(int num1, int num2){
    printf("%d and %d\n", num1, num2);
}
```

ex. return types

```
int myFunc() {
    int num1;
    printf("Enter a number");
    scanf("%d", &num1);

    return num1;
}
```

functions (syntax) examples

*note: function calls are intuitive

*IMPORTANT: must declare a function before calling it.

Function Declaration & Prototyping

- before *calling* a function, you must first *declare* it, like so:

ex.

```
float area(int height, int width);   ← declaration

float result = area(2,4);
                ↑ function call
```

  - this declaration tells the compiler about the function and how to properly call it. Without this, assumptions about the function (about return type, parameters, etc.) may be made incorrectly, breaking the code.
  - this also increases performance. In other languages, such as Java, the compiler looks ahead for function definitions, causing the code to be read through multiple times. In declaring the function before calling, the compiler is able to work in one pass – only looking ahead when explicitly instructed to.

- Declarations are also called "prototypes", giving only needed information for function call without defining the body until later.
- if a function takes no arguments, the keyword "void" should be used in their place – especially so in a prototype.

ex.

either is acceptable

function definition

```
int favNum() {
    return 7;
}
```

OR

```
int favNum(void){
    return 7;
}
```

declaration/prototype

```
int favNum(void);
```

Arrays

General Structure

```
type name[size]
```

- type: data type (of elements it will store)
- name: for reference
- size: length of array (how many elements) *must be CONSTANT

ex.

```
int ages[4];
```

just a declaration

```
int ages[4] = {19, 21, 24, 20};
```

also initialization

- unlike "lists" in other languages, the size of an array cannot change.
  → in order to "resize" an array in C, you must create an entirely new one, and scrap the original.

- to initialize an array full of zeroes, do this:

```
int name[size] = {0};
```

ex.

```
int arr[5] = {0};
```

↳ produces: {0,0,0,0,0}

- also works with partial fill:

```
int arr[3] = {2};
```

↳ produces: {2,0,0}

```
int arr[4] = {2,4};
```

↳ produces: {2,4,0,0}

Access

- Accessing an array is typical — uses square brackets

```
int arr[3] = {6,5,4};
```

indices: 0 1 2

arr[0] → results in 6
arr[1] → results in 5
arr[2] → results in 4

Arrays (Cont.)

- using a variable for the array size will throw an error without careful memory management.
  → an alternative to using a standard variable is to define a CONSTANT, using #define syntax.

ex.

```
#define SIZE = 7
```

2D - Arrays

structure

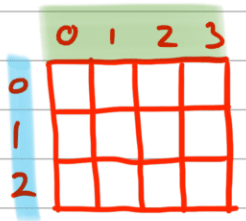
```
type name[size][size]
```

rows ↑ ↑ columns

ex.

```
int mat[3][4];
```

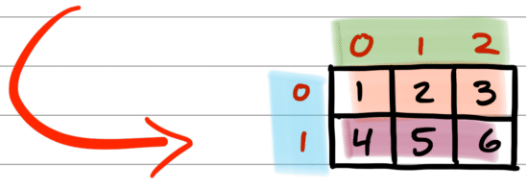
array of integers, 3 rows w/ 4 columns



ex.

```
int mat[2][3] = {{1,2,3}, {4,5,6}};
```

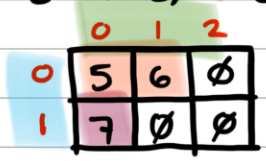
← standard initialization



ex.

```
int mat[2][3] = {{5,6}, {7}};
```

← "auto-fill" / incomple values method

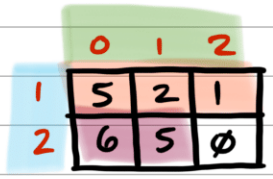


2D-Arrays (Cont.)

ex.

```
int mat[2][3] = {5,2,1,6,5};
```

lazy way (poor style)

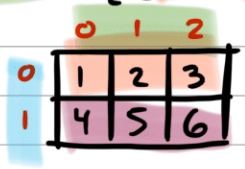


Access

- still typical - here are some examples.

```
int mat[2][3] = {{1,2,3}, {4,5,6}};
```

Array in-question:



ex.

`mat[0]` → produces first row
↳ in this case, {1,2,3}

`mat[0][1]` → intersection of first row & second column
↳ in this case, 2.

`mat[1][2]` → intersection of second row & third column
↳ in this case, 6.

Pointers

Why use Pointers?

- passing arguments without using pointers simply passes the value. Changes to those arguments once passed are local to the function and will not update externally. Using pointers, a reference to the original variable(s) is passed, allowing for changes within the function to reflect outside its scope.

- pointers pass the address of what they point to, rather than making a copy of them.

What are Pointers?

- a variable which stores/points to the address of some other data/variable.

Syntax & Usage

- declaration of a pointer uses the * symbol

ex.

```
int * p;
```
declares a pointer that will point to an integer

ex.

```
int a = 4;
int * p_to_a = &a;
```
(&a: the address of variable a)

creates a pointer (p_to_a) which points to the memory address of variable a.

ex.

```
*p_to_a += 1;
```
← this edits variable a directly, so printing the value of a would then return 5.

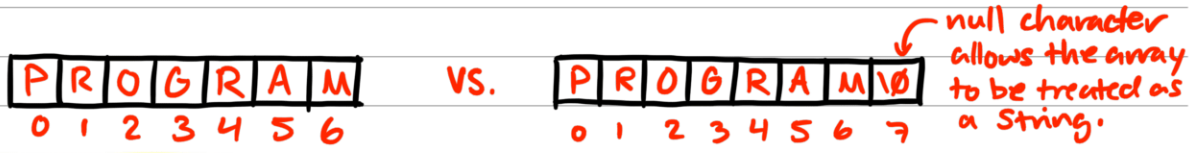
*Note: & = address (that's why we use it with scanf, to store the scanned value at the address of a given variable.)

Strings

Structure

- Strings in C are just arrays of chars.
- the last element of these arrays is always the termination (null) character : \0

eg.



*Strings in C are NOT their own datatype.

Initialization

Option A:

```
char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

critical component! otherwise not a string.

Option B:

```
char str[] = "Hello";
```

Input

notice no &! it's not necessary here.

```
scanf("%s", input);
```

restrict number of characters read:

```
scanf("%9s", input);
```

only stores the first 9 characters the user inputs, even if they put more.

- if string input has spaces, rather than scanf, use gets() function. scanf() reads until whitespace, gets() reads until newLine.

ex.

```
scanf("%s", input); → gets(input);
```

Output

Keep using the %s format specifier!

ex.

```
char name[] = "Jimmy";
printf("%s", name);
```

*could also use puts() function.

ex. puts(input);

*puts() adds a \n to the end automatically

<string.h> library

Notable Functionality

- `strlen(string);` returns length of string as size_t (unsigned integer type)

- `strcpy(*result, *string);` returns the copied string. Takes a destination (result) to copy to, and a source (string) to copy.
- `strcat(destination, source);` Takes two strings (destination and source) and concatenates them, storing the result in the destination.
- `strcmp(string1, string2);` Takes two strings. Returns 0 if equal. Returns >0 if string1 comes after, and <0 if string1 comes before.

*see: Library Documentation

Structures (Structs)

What is a struct?

- structures, in C, are similar to classes in other languages (such as java), with some key differences.
  - → notably, structs (or structures) in C cannot contain functions. They may contain references or pointers to functions, but no member functions of their own.

General Structure — "Structure Templates"

```
struct <structure name> {
    <datatype> <field name>;
};
```

keyword

create fields like normal variables

ex.

```
struct date {
    int day;
    int month;
    int year;
}
```

a struct definition for some "date" structure (or object)

Declaration

```
struct <structure name> <variable name>;
```

ex. `struct date myDateVar;`

Accessing Fields

dot notation

```
<variable name>.<field name> = value;
```

ex.
```
myDateVar.day = 1;
myDateVar.month += 2;
```

editing fields

Typedefs

- used for streamlining/simplifying code by defining a struct as its own (data) type
- removes the need for "struct" keyword when declaring a new variable as a struct.

Example:

```
typedef struct {
    int date, month, year;
} date;
```

fields

← struct name

Use:

```
date myDateVar2;
```

Structs (Cont.)

Initialization

- two main methods aside from a simple declaration without initialization.

Members in Order:

```
Date dt = {25, 12, 2022};
```

Designated Initializer:

```
Date dt = {.day=25, .month=12, .year=2022};   ← in order
```

OR

```
Date dt = {.month=12, .year=2022, .day=25};   ← out of order
```

OR

```
Date dt = {.year=2022};   ← other members initialized to zero.
```

Members / Fields

- fields (or members) of structs can be of any primitive type.
- fields can be arrays
- field types can even be other struct types!

Copying

- "value by value" copying of fields between two struct variables of the same type uses the = sign.

ex.

```
myDateVar1 = myDateVar2;
```

these two remain separate, but now have identical fields/members

- can use methods such as strcpy() (part of <string.h>) to copy values fully, so as to avoid a shallow copy — so that we are copying *values*, not *references*.

Relational Operators

- cannot use comparison operators such as < and > between two structs by default
  - → can define such comparisons to make it possible... sorta

Operator Overloading — not technically feasible in C

- just writing a logical function to replace relational operators

ex.

```
int equalPoints (Point p1, Point p2) {

};
```

Unions

What are they?

- basically structs, with some differing memory mgmt.
  - → structs clearly define each member (field) with their own chunk of memory. Unions, on the other hand, require all members to share allocated memory, making it so that only one member at a time can make use of that space.

- only one member can be initialized at a given point.

Use Case(s)?

- unions are useful when you have, for example, several properties/attributes that store information and you know you'll only use one, but you don't know which (until runtime).

Declaration

- same as structs — but union keyword

ex.

```
union numtypes {
    int intNum;
    float floatNum;
    double doubleNum;
};
```

union name

keyword

members. only one can be initialized at a time.

Accessing Members

- same as structs (dot notation)

Initializing Members

- same as structs — but only one.

Typedef

- same as structs

Dot vs. Arrow Operators

- when accessing members of structs or unions, we usually use the Dot operator (dot notation) like so:

ex.

```
struct Date myDate;
myDate.day = 2;
```

↑ dot operator

- when using pointers, structs' and unions' members are accessed using the Arrow operator, seen below:

ex.

```
union numTypes *myType;
myType -> intNum = 4;
```

pointer notation

↑ arrow operator

Constants

- not intended to be changed.
  → attempting to change value once set will throw errors.

Syntax

- uses const keyword

ex.

```
const int BIRTH_YEAR = 2001;
```

const: keyword
int: datatype
BIRTH_YEAR: constant name
2001: constant value

- if you do not define the datatype, type int is assumed.
- creating a pointer to a constant is valid. The constant cannot be changed/modified, but the pointer itself can be changed.
  ex. const int *ptr;

- can define a constant pointer (different from previous) where the pointer itself is constant & cannot change.
  ex. const int *const ptr;