A **data type** is:

- a set of possible values, and
- a set of possible operations on these values, and
- a representation of these values for a specific machine.

eg. a data type for integers

[-32768, +32767]

{+, -, *, /, %}

16 bits

Most imperative programming languages provide **basic data types** for:

- integers,
- floating-point numbers,
- characters,
- booleans, and possibly others.

The **C language** provides the **four basic types**:

- char
- int
- float
- double

and the **modifiers**:

- short 'at least 16 bits'
- long 'at least 32 (64?) bits'

- signed
- unsigned

The **C language** also provides **derived types** based on the **four basic types**:

- **pointers** (to entities of some type)

```
char* cptr;    // cptr holds the address of a char
```

- **arrays** (of elements of the same type)

```
char carray[5];  // carray is an array of chars
```

- **structs** (of members of possibly different types)

```
struct my_struct              int main()
{                             {
  int my_num;                   . . .
  char my_char;                 struct my_struct s1;
};                              . . .
```

- **unions** (of overlapping members of possibly different types)

```
union my_union                int main()
{                             {
  int my_num;                   . . .
  char my_char;                 union my_union u1;
};                              . . .
```

Pointers

```c
#include <stdio.h>

int main()
{
    int a, b;
    int* iptr;

    a = 2;
    iptr = &a;

    printf("a = %d\n", a);
    printf("iptr = %p\n", iptr);
    printf("&b = %p\n", &b);


    b = *iptr;
    printf("b = %d\n", b);

}
```

When the line below is used:

```c
printf("sizeof(iptr) = %lu\n", sizeof(iptr));
```

the ouput is:

```
sizeof(iptr) = 8
```

```
a = 2
iptr = 0x7ffeefbff53c
&b = 0x7ffeefbff538
b = 2
```

Pointers are not typically used in this way… we will see that they are really useful for manipulating dynamic (linked) data structures and for passing arguments to functions…

Arrays

An array is a contiguous sequence of elements of the same type.

```
#include <stdio.h>

int main()
{
    int a[5] = {2, 3, 7};

    printf("sizeof(a) = %lu\n", sizeof(a));
    printf("sizeof(a[0]) = %lu\n", sizeof(a[0]));

    printf("a[2] = %d\n", a[2]);
    printf(“*(a+2) = %d\n", *(a+2));
    printf("a = %p\n", a);
}
```

the array can be intialized with values

```
sizeof(a) = 20
sizeof(a[0]) = 4
a[2] = 7
*(a+2) = 7
a = 0x7ffeefbff520
```

Arrays ...strings

A string is an array of chars.

```c
#include <stdio.h>
#include <string.h>

int main()
{
    char first_name[20] = "Jack";
    char last_name[20];

    strcpy(last_name, "Black");

    printf("first name: %s\n", first_name);

    printf("initials: %c.%c.\n", first_name[0], *last_name);
}
```

```
first name: Jack
initials: J.B.
```

Multidimensional Arrays

```c
#include <stdio.h>

#define ROWS 3
#define COLS 3

double A[ROWS][COLS];
double B[ROWS][COLS];
double C[ROWS][ROWS];

void MM (double A[][COLS], double B[][COLS], double C[][COLS])
{
    for (int r = 0; r < ROWS; r++)
        for (int c = 0; c < COLS; c++)
            for (int k = 0; k < ROWS; k++)
                C[r][c] = C[r][c] + A[r][k] * B[k][c];
}

int main()
{
    // initialize matrices A and B
    for (int r = 0; r < ROWS; r++)
        for (int c = 0; c < COLS; c++)
        {
            A[r][c] = r+1.0;
            B[r][r] = 2.0;
        }

    MM( A, B, C );

    // print resulting matrix
    for (int r = 0; r < ROWS; r++)
    {
        for (int c = 0; c < COLS; c++)
            printf("%.2lf ", C[r][c]);
        printf("\n");
    }
}
```

← arrays are passed as pointers

output ?

structs

A struct is a contiguous set of members of possibly different types.

```c
#include <stdio.h>
#include <string.h>

struct student
{
    char first_name[30];
    char last_name[30];
    int final_grade;
};

int main()
{
    struct student s1;

    strcpy(s1.first_name, "Jack");
    strcpy(s1.last_name, "Black");
    s1.final_grade = 88;

    printf("name: %s %s\n", s1.first_name, s1.last_name);
    printf("grade: %d\n", s1.final_grade);
}
```

it's like we are creating a new type called `student`

we can access individual members with the ‘.’ operator

structs

A struct is a contiguous set of members of possibly different types.

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

struct student
{
    char first_name[30];
    char last_name[30];
    int final_grade;
};

int main()
{
    struct student s1;

    strcpy(s1.first_name, "Jack");
    strcpy(s1.last_name, "Black");
    s1.final_grade = 88;

    printf("name: %s %s\n", s1.first_name, s1.last_name);
    printf("grade: %d\n", s1.final_grade);
}
```

it’s like we are creating a new type called `student`

we can access individual members with the ‘.’ operator

```
name: Jack Black
grade: 88
```

structs

A struct is a contiguous set of members of possibly different types.

```c
#include <stdio.h>
#include <string.h>

struct student
{
    char first_name[30];
    char last_name[30];
    int final_grade;
};

int main()
{
    struct student s1;

    strcpy(s1.first_name, "Jack");
    strcpy(s1.last_name, "Black");
    s1.final_grade = 88;

    printf("name: %s %s\n", s1.first_name, s1.last_name);
    printf("grade: %d\n", s1.final_grade);

    printf("sizeof(s1) = %lu\n", sizeof(s1));
}
```

it's like we are creating a new type called `student`

we can access individual members with the '.' operator

```
name: Jack Black
grade: 88
?
```

unions

A union is a set of overlapping members of possibly different types.

```
#include <stdio.h>

union HW_Register
{
    struct Bytes
    {
        unsigned char byte0;
        unsigned char byte1;
        unsigned char byte2;
        unsigned char byte3;
    } bytes;

    unsigned int word;
};

int main()
{
    union HW_Register reg;

    reg.word = 0x12345678;
    reg.bytes.byte2 = 0xFF;

    printf("reg.word = %x\n", reg.word);

    printf("sizeof(reg) = %lu\n", sizeof(reg));
}
```

a single variable, i.e., the same memory location, can be used to store multiple types of data

unions

A union is a set of overlapping members of possibly different types.

```c
#include <stdio.h>

union HW_Register
{
    struct Bytes
    {
        unsigned char byte0;
        unsigned char byte1;
        unsigned char byte2;
        unsigned char byte3;
    } bytes;

    unsigned int word;
};

int main()
{
    union HW_Register reg;

    reg.word = 0x12345678;
    reg.bytes.byte2 = 0xFF;

    printf("reg.word = %x\n", reg.word);

    printf("sizeof(reg) = %lu\n", sizeof(reg));
}
```

a single variable, i.e., the same memory location, can be used to store multiple types of data

```
reg.word = 12ff5678
sizeof(reg) = 4
```

enums

An enum assigns names to integers for the purpose of readability and extensibilty.

```
#include <stdio.h>

enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum day today;

    today = tuesday;
    today++;

    if (today == tuesday)
        printf(“See you next Tuesday!\n”);
    else if (today == wednesday)
        printf(“See you next Wednesday!\n”);
}
```

```
See you next Wednesday!
```

typedefs

A typedef is used to create a name (alias) for another data type. It is often used to simplify the syntax of declaring structs and unions.

```c
#include <stdio.h>
#include <string.h>

typedef struct
{
    char first_name[30];
    char last_name[30];
    int final_grade;
} student;

int main()
{
    student s1;

    strcpy(s1.first_name, "Jack");
    strcpy(s1.last_name, "Black");
    s1.final_grade = 88;

    printf("name: %s %s\n", s1.first_name, s1.last_name);
    printf("grade: %d\n", s1.final_grade);

    printf("sizeof(s1) = %lu\n", sizeof(s1));
}
```

```c
typedef enum {F = 0,T = 1} Bool;

void PlayGuessingGame()
{
    // Generate a random number between 10 and 100 and find its square root
    srand((unsigned int)time(NULL)); // Seed rand with current time
    int numberToGuess = rand() % 91 + 10;
    double squareRoot = sqrt(numberToGuess);

    printf("%.8f is the square root of what number?", squareRoot);

    Bool done = F;

    while (!done)
    {
        int guess = GetGuess();

        if (guess < numberToGuess)
            printf("Too low, guess again: ");
        else if (guess > numberToGuess)
            printf("Too high, guess again: ");
        else
            done = T;
    }
    printf("You got it, baby!\n");
}
```

```
#define NUM_ROWS 5
#define NUM_COLS 5

typedef enum {F = 0,T = 1} Bool;

int GetInitialState(int init_state[])
{ can use strtok . . .}

void SetInitialState(char board[][NUM_COLS], int init_state[], int num_alive)
{. . .}

int CountLiveNeighbors(char board[][NUM_COLS], int r, int c)
{. . .}

void NextGeneration(char board1[][NUM_COLS])
{ may need another 2-D array to put results and then copy back }

void PrintBoard(char board[][NUM_COLS])
{. . .}

void PlayGameOfLife()
{
    // declare and init empty board
    // get and set initial state
    // while loop (call NextGeneration and PrintBoard)
}

int main()
{
    same as previous assignment
}
```