

Brief Intro to C

C Overview

- A typed language: the type of variable or expression determines how it is interpreted by the compiler. Casting gives the programmer control.
- Execution begins at the start of the main function, which takes two arguments: `int argc`, `char *argv[]`.
- Explicitly memory management: allocation (`malloc`), maintenance, and deallocation (`free`).
- A pointer is an address, pointers and arrays are the same data type.
- The symbol `&` means “address of”
- C syntax was the basis for Java, so basic control flow and assignment are the same
- String in C is an array of `char` end with `NULL`, represented by `\0`.
- C programs need to compile to generate an executable file. `gcc -o`
- `sizeof()`: return the size of a variable

```
/**
 * File: File Name followed by a short description of the program
 * Author: Your Name
 * Date: MM/DD/YYYY
 */

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

/* A short description of the function, if necessary
 */
int main (int args, char *argv[]) {

    int courseNum = 333;

    printf("Welcome to CS %d! \n", courseNum);

    // Integer types
    char    c1;
    unsigned char c2;
    short int integer1;
    int      integer2;
    long int integer3;

    // Float types
    float f1;
    double f2;

    // Pointer types
    unsigned char *ptr;
    ptr = (unsigned char *)&c1; // the symbol & means "address of"
    printf("The pointer points to an address with %u in it.\n", *ptr);

    // malloc: initialize pointers with memory
    ptr = (unsigned char *) malloc(sizeof(unsigned char));
    ptr = (unsigned char *) malloc(sizeof(*ptr));

    // free: return memory
    free(ptr);

    return 0;
}
```

Array

- Can stored in the stack or stored in the heap using malloc
- Can access array elements using index or pointer

```
/**
 * Array in C
 * array.c
 */

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

int main () {
    int i;
    int a[5];

    for (i = 0; i < 5; i++) {
        a[i] = i * 10;
    }

    for (i = 0; i < 5; i++) {
        printf("%d ", a[i]);
    }

    printf("\n");

    int *b;
    b = malloc(5 * sizeof(int));

    for (i = 0; i < 5; i++) {
        *(b+i) = i * 100;
    }

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

    free(b);

    return 0;
}
```

String

- String in C is an array of char
- String in C terminates with NULL “\0”
- To get the size of a string, we can use sizeof(string)/sizeof(char)
- string.h provides more functions for strings
 - int strcmp(const *str1, const *str2)
 - < 0: the first different char in str1 is less than str2
 - = 0: strs are the same
 - > 0: the first different char in str1 is larger than str2
 - char* strcpy(char *dest, const char* src)
 - copy the src to the dest
 - to avoid overflow, the size of dest should be large enough to contain the whole src string including the terminator NULL

```
/**
 * Print the whole string including the end character
 * str.c
 */

#include <stdio.h>

int main (int argc, char *argv[]) {
    char string[] = "Hello world!";

    int i = sizeof(string)/sizeof(char);
    printf("size of the string: %d\n", i);

    for (int j = 0; j < i; j++) {
        printf("%d ", string[j]);
    }
    printf("\n");

    return 0;
}

/**
 * Print the whole string without calculating its size
 * str2.c
 */

#include <stdio.h>

int main (int argc, char *argv[]) {
    char string[] = "Hello world!";

    for(int j = 0; string[j] != '\0'; j++) {
        printf("%c ", string[j]);
    }
    printf("\n");

    return 0;
}

/**
 * Copy a string
 * stringcpy.c
 */

#include <stdio.h>

int main (int argc, char *argv[]) {
    char string1[] = "Hello world!";
    char string2[20];

    int i = sizeof(string1)/sizeof(char);
    for (int j = 0; j < i; j++) {
        string2[j] = string1[j];
    }

    for (int j = 0; j < i; j++) {
        printf("%c ", string2[j]);
    }
    printf("\n");

    return 0;
}
```

```
/**
 * Copy a string using strcpy
 * strcpy2.c
 */

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

int main (int argc, char *argv[]) {
    char string1[] = "Hello world!";
    char string2[20];

    strcpy(string2, string1);

    int i = sizeof(string1)/sizeof(char);
    for (int j = 0; j < i; j++) {
        printf("%c ", string2[j]);
    }
    printf("\n");

    return 0;
}
```

```
/**
 * Compare two string using strcmp
 * strcmp.c
 */

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

int main (int argc, char *argv[]) {
    char str1[] = "this is";
    char str2[] = "a test.";

    if (strcmp(str1, str2) == 0) {
        printf("Strings are equal. \n");
    }
    else {
        printf("Strings are different. \n");
    }

    return 0;
}
```

Struct

- like the class in java, but without member functions
- it is a data type that can enclose multiple data types
- way to define a struct

```
struct Tag { // Tag is the name of the struct
    member 1; // members can be basic data types or other structs
    member 2;
    ...
};
```

- to access a member, use .

```
/**
 * Example of structure
 * structexp.c
 */

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

struct people {
    char name[25];
    char gender; //F or M
    char bday[8]; //YYYYMMDD, this is a bug, size should be 9, \0
    char address[50];
};

int main (int argc, char *argv[]) {

    struct people person;

    /*initialize student1*/
    strcpy(person.name, "Sherlock Holmes");
    person.gender = 'M';
    strcpy(person.bday, "18540106");
    strcpy(person.address, "221B Baker Street, London");

    /*print student1 info*/
    printf("%s: %c, %s, %s\n",
        person.name,
        person.gender,
        person.bday,
        person.address);

    return 0;
}
```

- What is the bug in this program? [\[bday\[8\] should be bday\[9\]\]](#)

Typedef

- used to create a synonym for an existing data type

```

/**
 * typedef example
 * typedef is used to create a synonym for an existing data type
 * synonym.c
 */

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

typedef unsigned int size_i;

typedef struct Mypuppy {
    char name[100]; // changing 100 to 99 won't impact the struct size why?
    [padding]
    int year;
    float weight;
} dog;

int main () {
    size_i a = 1;

    printf("size of a is %lu, value is %d\n", sizeof(size_i), a);
    printf("size of dog: %lu\n", sizeof(dog));

    dog* toto = (dog *) malloc (sizeof(dog));
    strcpy(toto->name, "toto");
    toto->year = 3;
    toto->weight = 25;
    printf("Name | Year | Weight\n");
    printf("%s | %d | %f \n", toto->name, toto->year, toto->weight);

    free(toto);

    return 0;
}

/**
 * More interesting structure
 * structure.c
 */

#include <stdio.h>

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

typedef struct {
    char* name;
    double price;
    date bestbefore;
} product;

int main () {
    product prod;
    prod.name = "product";
    prod.bestbefore.day = 1;
    prod.bestbefore.month = 6;
    prod.bestbefore.year = 2018;
    prod.price = 3.5;
    printf("%s, %d/%d/%d, %.2f\n", prod.name, prod.bestbefore.month,
        prod.bestbefore.day, prod.bestbefore.year, prod.price+2);

    return 0;
}

```

```

/**
 * Declare two mutually dependent
structures
 * mutualstruct.c
 * Compile won't work as it cannot decide
the size of struct B
 */

struct B;

typedef struct A {
    struct B b;
} A;

typedef struct B {
    struct A a;
} B;

int main () {
    A a;
    B b;

    return 0;
}

```

```

/**
 * Difference between char and string.
 * character.c
 */

#include <stdio.h>

int main () {
    char c1 = 'a';
    char c2 = "b";

    printf("c1 = %c, c2 = %c\n", c1,
c2); //unknow character for c2
    //printf("c1 = %c, c2 = %s\n", c1,
c2); //segment fault

    return 0;
}

```

```

/**
 * Declare two mutually dependent structures
 * mutualstruct2.c
 * This version works. As the compiler can
decide the pointer size
 */

struct A {
    struct B *b;
};

struct B {
    struct A *a;
};

int main () {
    struct A a;
    struct B b;

    a.b = &b;
    b.a = &a;

    return 0;
}

```

```

/**
 * Declare two mutually dependent structures
 * mutualstruct3.c (wrong, due to typedef)
 */

typedef struct {
    B *b;
} A;

typedef struct {
    A *a;
} B;

int main () {
    A a;
    B b;

    a.b = &b;
    b.a = &a;

    return 0;
}

```

Compile time error: errors detected when translating the program into machine language (syntax errors, type checking errors, compiler crashes(rarely)). If succeeds, meaning program.

Runtime error: happen when running the program. division by zero, dereferencing a null pointer, running out of memory, open an non-existed file

Bus error: fault raised by hardware, notifying an OS that process is trying to access memory that the CPU cannot physically address, or caused by alignment issue (read a long from an address which isn't a multiple of 4)

Segment fault: access memory that does not belong to your process (using a pointer to something was deallocated, using an uninitialized pointer, using a null pointer, overflowing a buffer)