day-4-functions-and-call-semantics

September 6, 2023

1 Day 4 Functions and call semantics

1.0.1 Topics

- functions & call semantics
- recursion

1. Functions & call semantics 1.1. Call semantics

1.1.a. Run the code cell below and examine the output. Can you explain the results?

Questions:

- When `process` is called, where is the variable `int b` allocated? What about the variable `int a` in `main`?
- When `b` is modified in `process`, what happens? Is the value of `a` in `main` modified? Why?
- What are the addresses of `a` and `b`? Are they different? Why? *Hint: use `printf("&x=%016p\n", &x)` to print the address of x in hexadecimal format.*
- What happens to variables local to `process` when the function returns?

[ ]:
```c
//// process-value.c
#include <stdio.h>

void process(int b) {
    printf("start process: b=%d\n", b);
    b += 99;
    printf("end process: b=%d\n", b);
}

int main(void) {
    int a=5;
    printf("start main: a=%d\n", a);
    process(a);
    process(++a);
    process(a++);
    printf("end main: a=%d\n", a);
    return 0;
}
```

1.1.b. Consider the code below. What is wrong with the function `get_address`?

[ ]:
```c
//// bad_func.c
#include <stdio.h>

int *get_address(void) {
    int i = 42;
    return &i;
}

int main(void) {
    int *q = get_address();
    printf("q = %016p\n", q);
    printf("*q = %d\n", *q);
    return 0;
}
```

1.1.c. Run the code cell below and examine the output. Can you explain the results?

Questions:

- What is the type of `b` in `process`? Is this type the same as or different to `a` in `main`?
- Where is the variable `b` in `process` allocated?
- What is the meaning of `&a` when calling `process(&a)`? How is this different from `process(a)` above?

[ ]:
```c
//// process-pointer.c
#include <stdio.h>

void process(int *b) {
    printf("start process: *b=%d\n", *b);
    *b += 99;
    printf("end process: *b=%d\n", *b);
}

int main(void) {
    int a=5;
    printf("start main: a=%d\n", a);
    process(&a);
    printf("end main: a=%d\n", a);
    return 0;
}
```

1.1.d. In the lecture we saw some (not-working) code to swap two variables. Write a short C program which swaps two values.

[ ]:
```c
//// swap.c

int main(void) {
    int p = 0, q = 5;
    swap(/* what are the parameters? */);
```

```
    // p and q should be swapped here
    return 0;
}
```

1.2. Declaring and defining functions

Complete the program below, defining functions with the signatures `char *get_user_name(void)` and `int get_user_age(void)` which get the user's name and age from stdin and return the results. Test your function with a few different inputs.

[ ]:
```
//// get-user-name.c
#include <stdio.h>

int main(void) {
    char *name = get_user_name();
    int age = get_user_age();
    printf("user %s is %d years old\n", name, age);
    return 0;
}
```

1.3. (challenging) Returning results via pointer arguments

(challenging) In C, it is only possible to `return` one value from a function. However, if we want a function to provide multiple results we can achieve this by passing arguments as pointers instead of by value. Consider this function:

```
void write_value(int *value) {
    *value = 42;
}
```

The function `write_value` accepts one `int *` and writes the value `42` into the `int` whose address is stored in `value`. It is used like so:

```
int quantity = 0;
write_value(&quantity); // quantity is now 42
```

When writing functions which write data out through pointer arguments, it is good practice to gracefully handle a `NULL` pointer. It is never valid to dereference a `NULL` pointer.

Complete the program below, defining a function with the signature `void get_user_name_age(char **name, int *age)` which gets the user's name and age from stdin and writes the results out to the given pointers. Test your function with a few different inputs.

How could this function be made to fail? How can you guard against that?

[ ]:
```
//// get-user-name-age.c
#include <stdio.h>

int main(void) {
    char *name = NULL;
    int age = -1;
```

```
    get_user_name_age(/* what are the parameters? */);
    printf("user %s is %d years old\n", name, age);
    return 0;
}
```

1.4

Write a 3rd version of this program, but this time use a suitably defined `struct User` to hold a user's name and age. Define a function `struct User get_user(void)` to obtain input from the user. Define a 2nd function `void get_user_ref(struct User *user)` which writes the result out using the provided `struct User` pointer argument.

What happens if an invalid pointer value is passed to `get_user_ref` at runtime? What could your program do to handle this?

[ ]:
```
//// get-user-name-age-struct.c
#include <stdio.h>

struct User {
    // define me!
};

struct User get_user(void);
void get_user_ref(struct User *user);

int main(void) {
    struct User user1 = get_user();
    printf("user %s is %d years old\n", /* ... */);
    return 0;
}

struct User get_user(void) {
    // return a struct User obtained from user input
}
void get_user_ref(struct User *user) {
    // return a struct User via the pointer argument
}
```

2. Recursion A recursive function is one which calls itself. To make a function recursive, it must handle:

- the base case (the computation which stops further recursion)
- the general case (the computation expressed in terms of a smaller, recursive computation)

What would be the consequence of a recursive function which had no base case?

Let's explore recursion by implementing a few functions with and without recursion.

2.1

In the code cell below, write a function `int sum_to(int n)` which sums all the positive integers up to `n`. Test it on a few different inputs.

[ ]:
```c
//// sum_to_n.c
#include <stdio.h>

int main(void) {
    int n=5;
    printf("sum to %d: %d\n", n, sum_to(n));
    return 0;
}
```

Re-implement the function `sum_to` using recursion and check your results against the non-recursive version.

What is the base case for this recursive function? What is the general case?

[ ]:
```c
//// recursive_sum_to_n.c
#include <stdio.h>

int main(void) {
    int n=5;
    printf("sum to %d: %d\n", n, sum_to(n));
    return 0;
}
```

2.2

Implement a recursive function with the signature `int str_length(const char *s)`. It should return the number of characters in the string `s`, excluding the terminating null byte `'\0'`. You can assume that the input `s` is already null-terminated.

Test your function on a few string literal arguments.

[ ]:
```c
//// strlen_r.c
#include <stdio.h>

int main(void) {
    const char *country1 = "South Africa";
    const char *country2 = "United Kingdom";
    const char *country3 = "Japan";
    printf("str_length(\"%s\") = %d\n",country1, str_length(country1));
    return 0;
}
```

2.3

(challenging) Write a recursive function `int str_compare(const char *lhs, const char *rhs)` which compares the null-terminated strings `lhs` and `s2` alphabetically, returning:

- `-1` if `lhs` is before `rhs` alphabetically
- `0` if `lhs` and `rhs` are identical

- `+1` if `lhs` is after `rhs` alphabetically

Hints:

- When does the recursion need to terminate? Is it only when the end of a string is reached?

[ ]:
```c
//// str_compare.c
#include <stdio.h>

int str_compare(const char *lhs, const char *rhs) {
    return 0;
}

int main(void) {
    const char *s1 = "Durham";
    const char *s2 = "Edinburgh";
    const char *s3 = "Newcastle";
    const char *s4 = "Dirac";
    const char *s5 = "Dirac STFC";
    printf("str_compare(\"%s\", \"%s\") is: %d", s1, s2, str_compare(s1, s2));
    return 0;
}
```