C Programming – Part 3

Aravind Prakash

aprakash@Binghamton.edu

Contents

Contents

C Preprocessor

- Begins with ‘#’. Example #include<stdio.h>
- #include, #define, #undef, #if, #ifdef, #ifndef #error
- Preprocessing is one of the first steps during compilation.
- Try the –E option with gcc to see the preprocessed output.
- Ref: http://www.cprogramming.com/reference/preprocessor/

Macros

- #define something something_else
- Every occurrence of something is replaced by something_else. This is a blind search and replace. There is no check for syntactic correctness.
- Example:
  - #define sum(a,b) a+b
  - #defind diff(a,b) a-b

Macros

- #define something something_else
- Every occurrence of something is replaced by something_else. This is a blind search and replace. There is no check for syntactic correctness.
- Example:
  - #define sum(a,b) a+b
  - #defind diff(a,b) a-b /* WRONG: Will not work!!! */

Macros

- #define something something_else
- Every occurrence of something is replaced by something_else. This is a blind search and replace. There is no check for syntactic correctness.
- Example:
  - #define sum(a,b) a+b
  - #defind diff(a,b) a-b /* WRONG: Will not work!!! */
    - diff(3-1, 2-5) != 3-1-2-5

Macros

- #define something something_else
- Every occurrence of something is replaced by something_else. This is a blind search and replace. There is no check for syntactic correctness.
- Example:
  - #define sum(a,b) a+b
  - #defind diff(a,b) ((a)-(b)) /* Correct! */
    - diff(3-1, 2-5) == ((3-1)-(2-5))

Contents

Call by value

- Arguments are copied from the caller to callee.
  - E.g.,

```
void foo(int x) { printf("%d\n", ++x); }
int main() {int i; i = 10; foo(i); printf("%d\n", i);}
```

- Callee gets its own copy of the argument. That is, x is a local variable in foo. The scope rules apply. So, any changes to x in foo are not reflected in caller.

Contents

- C Preprocessor and Macros
- Functions: Call by value
- Functions: Call by reference
- Calling convention (x86 and x86_64)
- Introduction to pointers

Call by reference

- A reference to (or the address of) the argument is passed to the callee from the caller.
  - E.g 1

    ```
    void foo(int x[]) { printf(“%d\n”, ++(x[0])); }
    int main() {int i[1]; i[0] = 10; foo(i); printf(“%d\n”, i[0]);}
    ```

    E.g. 2

    ```
    void foo(int *x) {printf(“%d\n”, ++(*x));}
    int main() { int i; i = 10; foo(&i); printf(“%d\n”, i); }
    ```
- Arrays are always passed by reference. Changes made in the callee persist.

Contents

- C Preprocessor and Macros
- Functions: Call by value
- Functions: Call by reference
- Calling convention (x86 and x86_64)
- Introduction to pointers

Calling convention

- An agreement between the caller and the callee on:
  - How and where arguments are going to appear
  - How and where return values are going to appear
  - Alignment, who cleans-up, etc.
- In x86 (32bit), all arguments go on the stack. Arguments are pushed from right to left. That is, last argument is pushed first, then the one before last and so on. First argument is pushed last.
- In x86 (64bit), first few arguments go in the registers rdi, rsi, rdx, rcx, r8 and r9. Remaining go on the stack similar to x86-32.

Calling convention

- Return values are passed in eax (x86) or rax (x86_64) registers.
- Caller cleans up the stack (i.e., remove arguments from the stack) when the callee returns on most linux implementations.
- Callee is responsible for allocating and deallocating whatever stack space it needs.
- More here: https://en.wikipedia.org/wiki/X86_calling_conventions

Contents

Introduction to Pointers

- A pointer ``points" to memory.
- A pointer is preceded by `*' in its declaration. (e.g., int *p;)
- `*' is also used to dereference a pointer. That is, it is used to retrieve the value the pointer points to. (e.g., x = *p;)
- Type of pointer specifies the type of memory that the pointer points to.
  - E.g., int *p; /* p is a pointer to type int */
- Address of a variable is obtained through '&' operator. It is called the reference to a variable. It can be stored into a pointer. (e.g., int *p; int x; x = 10; p = &x;).
  - Try samples at: https://cdecl.org/
- Pointers and reference are useful for call-by-reference.

Introduction to Pointers

- Pointers can also point to other pointers.
  - E.g., int **p; /* p is a pointer to pointer to int */
- Pointers can also be generic, i.e., void pointers.
  - E.g., void *p; /* p can point to any type */
- Void pointers can not be dereferenced because dereferencing requires knowledge of the type being dereferenced.

Casting

- When one type is converted (or interpreted) as another type, it is called casting.
- Void pointers are often casted before use. (e.g., void *p; char *c; c = (char *) p)
  - Here, you are telling the compiler: ``treat p as a pointer to character”.
- In general (x) y is treating y as a type x.

Casting

- When one type is converted (or interpreted) as another type, it is called casting.
- Void pointers are often casted before use. (e.g., void *p; char *c; c = (char *) p)
  - Here, you are telling the compiler: ``treat p as a pointer to character”.
- In general (x) y is treating y as a type x.