C Basics 9: Functions and Parameter Passing

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling

C Basics 9: Functions and Parameter Passing

9. **Functions and Parameter Passing**
   - **defining functions**
   - **functions vs. procedures**
   - **parameter passing: call-by-value**
   - **array parameters**
   - **pointer parameters**
   - **pointers as reference parameters**
   - **const type modifier**
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Miscellaneous

Defining Functions

Functions are the the main code/algorithm modules in C, i.e., **subroutines**.

C *functions* are similar to **methods** in Java, except that functions are *not* members of classes.

The basic syntax for a function definition is:
return_type function_name (*parameter_list*) {*function_body*}

- *return_type* must be a *non-array* object type or else `void`
- *function_name* must adhere to naming rules for *identifiers*
- *parameter_list* is either `void` or a *comma-separated list* of *parameter_declaration*'s
- a *parameter_declaration* specifies the parameter type and its name, similar to variable declarations

Functions vs. Procedures

C functions may have a `void` return type, meaning they do not return an object/value.

Such functions are frequently referred to as **procedures** in C literature.

However, the term *"procedure"* is not used in recent C standards!

Functions and Arrays and Pointers

C functions do *not* allow (copies of) *arrays* to be passed:

- array *parameters* are automatically converted to equivalent pointer types
- array *return values* are *not* allowed (but pointer types are)

Because of the connection between arrays and pointers, these restrictions have little practical effect.

They do however mean that *array-type parameters* can leave the array size *unspecified*:

- `int x[10]` (size specified as 10, but not bounds checked)
- `int x[]` (size unspecified)
- `int x[*]` (size unspecified, *variable-length array*)

(With *multidimensional array* parameters, only the *leftmost dimension* can be left unspecified, e.g., `int x[][20][30]`.)

Parameter Passing

C uses a **call-by-value** scheme for passing function parameters: parameters are bound to *copies of the values* of the arguments.

This is the same scheme used by Java and most other modern programming languages.

The key effect of this scheme is that argument variables will not have their values changed by a function, only the local (copy) values change.

Since an array parameter is convereted to a pointer, however, an array parameter is bound to a copy of the argument array *address*, so array elements *can be modified* inside of a function.

This also makes it *efficient* to use arrays as function parameters, because the array itself is not copied.

Parameter Passing (contd.)

Consider the following version of `swap()`:

```
void swap(int x, int y)
{
  int temp = x;
  x=y;
  y=temp;
}
```

This does not work:

```
int i=1, j=2;
swap(i,j);
printf("i:%d  j:%d\n",i,j);  //Oops: still i==1, j==2
```

Parameter Passing (contd.)

In C, the effect of **call-by-reference** can be achieved by using *pointer parameters*:

```
void swap(int *x, int *y)
{
  int temp = *x;
  *x=*y;
  *y=temp;
}
```

Now `swap()` works:

```
int i=1, j=2;
swap(&i,&j);
printf("i:%d  j:%d\n",i,j);  //Now i==2, j==1
```

Parameter Passing (contd.)

Using *pointer types* as *function parameters* is very common, but leads to confusion about where memory is being allocated.

For example, the `wait()` system call takes an `int*` parameter because it will change the value of its argument, but an `int` must have been pre-allocated:

```
int status;
wait(&status);
```

On ther other hand, a function like `char *getline(FILE *fptr)` could create and return a string, so we can simply do:

```
char *line = getline(fpntr);
```

const

const is a **keyword** that serves as a **type qualifier**.

This means that it can be added to a type declaration:

```
const int x = 10;
```

Adding the `const` qualifier tells the compiler that a variable (or parameter) is *not to have its value modified* (after it is initialized in the declaration).

If the code attempts to modify the value of a `const` variable, the compiler will generate an *error message*:

```
x = 20;  //This line will cause the compiler to complain!
```

const (contd.)

The only confusing thing about `const` is its use with *pointers*:

- `const int *ip = &i;`
  - `ip` is of type *pointer to const-qualified int*
  - `ip` can be modified (made to point to another `int`), but what it points to cannot be changed (using `ip`)
- `int * const ip = &i;`
  - `ip` is of type *const-qualified pointer to int*
  - `ip` cannot be modified (will always point to `i`), but what it points to (`i`) can be changed
- `const int * const ip = &i;`
  - `ip` is of type *const-qualified pointer to const-qualified int*
  - neither `ip` nor what it points to can be modified