

# CSIS0230A Principle of Operating Systems (Class A)

## Notes for Tutorial 1

### C++ sweN

In the first year programming courses, you familiarized yourselves with a programming language called C++. With its many advanced features, it serves very well as a convenient and extensible system-level programming language, suitable for most low to medium level applications.

Unluckily, the convenience and extensibility features of C++ do not come with no cost. They increase the size of all executable programs, make programs execute a bit slower, and make the compiler more complicated. The Linux kernel, with its high demand on performance and resources utilization, does not use C++. It uses C instead. Furthermore, many of the C library functions are unusable from within the kernel.

It is for this reason that we have to examine the “reverse News” (you see why the title of our notes have such strange capitalization, right?) of the C++ language: We will be doing quite some C programming during this course.

## 1. Hello world

The traditional hello world in C is actually both a C and C++ program. Indeed, most C programs can be compiled by a C++ compiler right away.

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

You should immediately notice the similarity with C++. The differences:

- We include `stdio.h` instead of `iostream`. C uses a much simpler system for printing. “Simpler” here is for the compiler and the run-time library: using it is more cumbersome than using the C++ library. Accordingly, printing is done not using the object `cout`, but the function `printf`. Also notice that we have no `endl` to print. We must write `"\n"`.
- The C++ namespace support is missing. We never use a namespace or use the `::` operator in C. In fact, class support is also missing, so `::` is really not a C operator.

## 2. Where are all those variables defined?

The differences between C and C++ do not stop with those illustrated by hello world.

One notable difference is the statement ordering. In C++, **variable declaration statements** can appear anywhere within a block. In C, they **can only appear at the beginning of a block**.

This is not as drastic as it might seem. Since there is no class in C, there is no “initialization code” (constructors) that needs to be executed when a variable is declared. Also, variables can be defined at the beginning of nested blocks as well. Many C programmers separate the declaration statements from the other statements by a blank line, so that they can quickly find where to add new variables.

In C++, **variables** can be declared **within the parentheses of a if, for or while statement**. It is

heavily used to make loop variables. For example,

```
int sum = 0;
for (int i = 0; i < 10; ++i) // Not in C!
    sum += i;
```

This **is not allowed in C**. So we have to rewrite the above so that *i* becomes a local variable, e.g., declared with *sum*. This can result in clashing variable names, so be careful.

### 3. Same name for different functions?

In C++, we can define functions within structures or classes, so that we can call the function for a structure instance (i.e., object), passing a pointer to itself automatically. Such mechanism is convenient because it allows name overloading: different structures can define different functions of the same name, and the C++ compiler always selects the right one to call.

Such **overloading facility is not present in C**. In fact, even simple function overloading is not allowed: unlike C++, **there must not be two or more functions with the same name**, even if they have different signatures (argument length or types).

With the lack of type-based overloading, the type system of C is in general more sloppy than C++. For example, a literal char is an integer (as well as all enum constants):

```
#include <stdio.h>
int main() {
    char a;
    printf("%d\n", sizeof('a')); /* 1 in C++, 4 in C */
    printf("%d\n", sizeof(a)); /* 1 in both C and C++ */
    return 0;
}
```

### 4. My recursive data types fails!

At the beginning of a struct definition, right after the keyword **struct**, we can place an identifier for later referral. In C++, that definition is a type, and is considered to be defined immediately. In C, that definition is called a “**struct tag**”, and **is not considered to be the name of a type**. This can be very annoying. For example, the following C++ code cease to work:

```
struct list {
    int data;
    list *next; // Not in C!
};
```

Since *list* is not considered to be a type, *list \* next* is not a valid field. We write it as follows:

```
typedef struct list {
    int data;
    struct list *next; /* Okay in C */
} list;
```

Notice that **struct id** is always a type name for whatever *id*, even those undefined ones. *After* the above definition, *list* becomes the name of a type, thanks to **typedef**. But that type name is not

available within the **struct** definition.

Also notice that the C++ double-slash comment style is not available in C in general, even though gcc allows its use when not compiling with `-ansi`.

## 5. I can't remember that many things!

The C++ dynamic memory system is centered around the **new** and **delete** operators. Both are missing from C. Instead, **dynamic memory allocation are done using library functions**. The mostly used ones are *malloc*, *calloc* and *free*. Instead of having a lot of words to define them, let's see a C program side by side with a C++ one.

```
int main() {                                     #include <malloc.h>
    int *c = new int;
    int *carr = new int[10];
    // Do something with c and carr
    delete [] carr;
    delete c;
}

int main() {
    int *c = (int *)malloc(sizeof(int));
    int *carr = (int *)calloc(10,
    sizeof(int));
    /* Do something with c and carr */
    free(carr);
    free(c);
}
```

Apart from having different number of arguments, there is one more important difference between *malloc* and *calloc*: *calloc* will clear the allocated memory so that all bits are 0. Since the language does not know about dynamic memory allocation, there is no constructor or destructor that get called when the object is created or destroyed. The library only treats the allocated memory as a single block of memory. For example, *carr* is just a chunk of memory of 40 bytes. The library won't care that it is used to store 10 integers.

Such ignorance of the compiler and the library makes it necessary to cast the result of allocation to the required pointer type. Note that the casting operator is unlike the C++ casting operator: there is nothing called **static\_cast** or **dynamic\_cast**, since the type system of C is so simple.

Strictly speaking, the casting is not necessary, since C allows pointers of type **void \*** (returned by *malloc* and *calloc*) to coerce to any other type of pointers. But using this feature cause difficulties when compiling the program with C++, and is not recommended.

## 6. Is it a pointer or an array?

Unlike C++, **pointer arithmetics and pointer-array equivalence play an important role in C**. In particular, you will see a lot of pointer additions in C code. As an example, given a C-style string *f* (presumably a filename), the following will return the extension of it:

```
char *get_ext(char *f) {
    char *ret = NULL;
    while (*f) /* i.e., while string not ended */
        if (*f++ == '.') /* Check *f, and point f to the next char */
            ret = f; /* set ret to f if *f was '.' */
    return ret;
}
```

```
}
```

Note that *NULL*, not 0, is used for as the null pointer. It is defined in a number of libraries, but the standard way to get it is from `stdlib.h`.

## 7. This work exactly for all types

In C++, we have a facility called **template** which can be used to make functions automagically. Again, such facility is **missing in C**. Instead, **C uses a very rudimentary string-based system called macro expansion**. A macro can be function-like or constant-like. Let's just see an example:

```
#define PI 3.141592653589793
#define max(a, b) ((a)>(b)?(a):(b))

int main() {
    printf("%f\n", max(PI, 1/PI));
}
```

Note the large number of parentheses used in the macro, to make sure that precedence surprise can never happen. But there is usually one problem: the operands are evaluated twice instead of once. For example, if we call `max(f(), 2)`, `f()` is called. If it turns out to be larger than 2, `f()` is called again, possibly resulting in another value. It is definitely slower than using the old value again. For these reasons, macros are not usually used in C++.

## 8. What is provided by the system?

Many C functions (like *sleep*, *sqrt*, etc) are useful in C++ as well, so you probably know them already. The only precaution is that **if you use mathematical functions, remember to add `-lm` to the compilation options** when compiling, to link with the math library. It is automatically done in C++, but in C you need to do this manually.

However, since C++ replaced many C function by classes, especially STL ones, some of the functions in C are not this well known by C++ programmers. Examples are *qsort* and *bsearch*, which perform quick sort and binary search on an array.

As we have seen, the **input and output system of C is based on functions** instead of objects. The most important ones are *printf*, *puts*, *scanf* and *fgets*. There are file and string versions of *printf* and *scanf*, which prefix the function names by a character 'f' and 's' respectively. For example, to convert an integer to a string, we do this:

```
char s[20];
...
sprintf(s, "%d", num);
```

You probably have been wondering what are those %'s for quite a while. They are called **format strings** for C. For example, "`%d`" notify that an argument is an integer, to be printed (or readed in *scanf*) in decimal. Some other important format strings are `c` (for character), `s` (for string), `x` (for hexadecimal string), `f` (for fixed-point floats) and `e` (for scientific notation like `1e5`). Another useful feature is width specifier. E.g., `printf("%5d", 15)` will pad three spaces before 15 so that it occupies 5 characters, `printf("%6.2f", 2.3)` will print the string "2.30", which contains 2 digits

after the decimal point; and will print two spaces before that to make it 6 characters.

All standard C functions have their man pages. Depending on whether they are “system calls” (we will know what it means later in the course), they are documented in section 2 or 3 man pages. In particular, *printf* and *scanf* are not system calls. Thus they are documented in man pages of section 3. Type `man 3 printf` (or `man -s 3 printf` in Solaris) to read them.

## 9. We can't do OOP in C?

The primary plus of C++ over C is that it support object oriented programming directly, using **class** construct. With such a facility, **old code** in a class function **can call new code** defined in derived class. Such facility is missing in C.

Instead, C allows old code to call new code by using **function pointers**. The name of every function is a function pointer literal (constant). For example, if we define

```
double my_sqrt(double) { ... }
```

Then *my\_sqrt* is a function pointer of type **double**(\*)(**double**). That is, it points to a function that takes a **double** as argument and return a **double**. We can create variables or arguments of such function pointer type, or store them in struct fields, just like values of other types. About the only reasonable thing done for such pointers is to call the function that is pointed to. For example, the following function receives such a pointer and call it.

```
void callit(double (*)(double)) {  
    p(2.0);  
}
```

A caller would simply pass *my\_sqrt* to *callit*:

```
int main() {  
    callit(my_sqrt);  
}
```

This facility is very useful when writing library functions (like *qsort* and *bsearch*) and implementing object oriented programs in C. For each type, a **struct** containing such function pointers is defined, so that a derived type can override a function by using a different struct instance.

## Conclusion

While not entirely without hassle, it is possible for a C++ programmer to pick up C and write C programs. Of course, it is more cumbersome to write in C: that's why C++ is there. But we gain in speed and resources utilization, which is essential in OS kernel code.