

# CSIS0230A Principle of Operating Systems (Class A)

## Notes for Tutorial 1

### From C++ to C

With many advanced features of C++, it is a convenient and extensible system-level programming language suitable for most low to medium level applications. Unluckily, such features come with a cost: they increase the size and running time of executables, and make the compiler more complicated. Since the OS is used by every programs, efficiency and resource utilization is a first priority. As a result, Linux kernel doesn't use C++, but C instead.

C programs are pretty similar to C++ ones, and indeed most C programs can be compiled by a C++ compiler, including the traditional hello world program. We normally compile C source files using `gcc` instead of `g++`, so that the more bulky C++ library is not used. The file extensions of C programs are normally `.c` instead of `.cpp` or `.cc`.

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

You should realize that it is indeed very similar, except the `printf` function that you might not know. There are something that we write in C++ programs but won't write in C programs, like the "using namespace" declaration. In this following sections we will see what other things can't be done in C, and how to work around them. We will show code in C++ on the left and the corresponding code in C on the right to illustrate the ideas.

It is possible for a C++ programmer to pick up C and write C programs without too much trouble. 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. While the notes is long, just reading the illustration should give you a good idea about what it is about. You should be able to skim over most parts if time is scarce.

#### 1. Defining variables

<pre>int main() {     int sum = 0;     for (int i = 0; i &lt; 10; ++i)         sum += i;     double rootsum = sqrt(sum); }</pre>	<pre>int main() {     int sum = 0, i;     double rootsum;     for (i = 0; i &lt; 10; ++i)         sum += i;     rootsum = sqrt(sum); }</pre>
--	--

In C++, local variables can be defined nearly everywhere<sup>1</sup>. Until recently, in C they can only be defined **before executable statements in a block**. So everytime you write a `{`, you can write variable declarations, but once you write the first executable statement, you must not write declarations until another `{`. The original believe is that writing variables in the beginning of a block would make a program easier to read, but practice reveals that it is more error-prone. Newer C standard (C99) allows variables to be used everywhere, but it is not yet the default in

---

<sup>1</sup>Global variables are always allowed everywhere in both C and C++.

gcc. So using it will trigger the following error:

```
...: error: `for' loop initial declaration used outside C99 mode
```

Note that the above code makes *i* defined throughout the function. So be careful when using this technique—a variable cannot be defined twice.

## 2. Same name for different functions?

```
int sum(int a, int b) {
    return a+b;
}
double sum(double a, double b) {
    return a+b;
}

int sum_i(int a, int b) {
    return a+b;
}
double sum_d(double a, double b) {
    return a+b;
}
```

In C++, functions with different type of arguments can be of the same name. **In C, different names must be used for different functions.** The former program results in the following.

```
...: error: conflicting types for `sum'
...: error: previous declaration of `sum'
```

Operator overloading is also absent in C.

## 3. Functions without argument

```
int f1() {
    // code
}
int f2(...) {
    // code
}

int f1(void) {
    /* code */
}
int f2(...) { /* drop ... is equivalent */
    /* code */
}
```

If you don't write anything within the pair of parentheses of a function declaration or definition, in C++ it means the function takes no argument. In C, however, it means the function takes variable number of arguments (of unspecified types). **To mean a function without argument you must write void in the parentheses.**

If you forget this, the compiler will not catch problems like calling *f1(43)* (i.e., calling *f1* with too many argument), but usually the program will still compile. There is an exception: if you get a function pointer (see Section 11) of it, it will be of the wrong type, and the compilation fails.

## 4. Boolean values

```
bool finished = true;
if (a == 0)
    finished = false;
if (finished) {
    // other code
}

int finished = 1;
if (a == 0)
    finished = 0;
if (finished) {
    // other code
}
```

C++ has a type called **bool**, which is used to represent true or false values. C has no **bool**. Instead, **C uses any integer or pointer type to represent true-or-false values**. Anything non-zero and non-NULL are treated as true. (The interpretation also works for C++, but is not good style since it is more difficult to know that the variable only has 2 values.)

## 5. References and Pointers

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
void caller(void) {
    int a=2, b=3;
    swap(a, b);
}

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void caller() {
    int a=2, b=3;
    swap(&a, &b);
}
```

In C++, “reference arguments” can be used for functions to write to the memory of callers. C does not have reference variables, so the above code won’t compile, resulting in the following error (and a lot more).

```
...: error: syntax error before ‘&’ token
```

Instead, a C function that needs to write to a variable should receive a pointer to the variable as argument. This is a general mechanism, and *you must familiarize yourselves with it*. E.g., if you read the following function prototype in a manual page of a function,

```
pid_t wait(int *status);
```

it means that the function expects you to give it a pointer of integer. But you must not call it by simply creating a **int \*** variable:

```
int *status;
wait(status); /* definitely wrong: wait() would use memory at random location! */
```

If you compare it with the following you can see what’s wrong:

```
double f;
return sqrt(f);
```

*f* (and *status*) is not initialized, so it contains undefined value (for *status*, undefined address). Most of the time, one should instead create an integer variable and pass its address to the function:

```
int status;
wait(&status); /* correct usage */
```

Sometimes it makes more sense to allocate memory using *malloc* (see Section 7), but you should remember to free them. And sometimes the function allows you to pass it a *NULL* pointer, when the function is notified that you don’t use that argument. In this case, the function should be called like *wait(NULL)*.

## 6. Structure declarations

```
struct list {
    int data;
    list *next;
};

typedef struct list {
    int data;
    struct list *next;
} list_t;
```

Right after the **struct** keyword, a name is usually placed to identify the type of the struct. It is called a **struct tag**. In C++, struct tags are type. In C, **struct tags are not type**. To form a type **you must write struct name** where the *name* is the struct tag. The C++ code will thus produce the following strange error when the compiler doesn't understand what is *list*:

```
...: error: syntax error before "list"
```

Note that in the above example, *list\_t* is considered to be defined only after the **typedef** statement, so you can not replace **struct list** by *list\_t* within the definition. Outside, you can do so, which may be more convenient (but Linus, the author of Linux, doesn't like this, since it makes it less clear that *list\_t* is actually a structure type).

## 7. Dynamic memory

```
int main() {
    int *c = new int;
    int *carr = new int[10];

    // Do something with c and carr
    delete [] carr;
    delete c;
}

#include <malloc.h>
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);
}
```

C++ has language-level facilities for allocating memory dynamically (**new** and **delete**), so the compiler knows what is the types of values that are being created and destroyed, and call constructors and destructors accordingly. In C, the language provides no such facility, and it is instead provided by the standard library which knows nothing about types or constructors. As a result, all memory allocations are done by the following steps:

- Find the number of bytes of memory to allocate, typically by finding the size of each data element and multiply it by the number of copies needed.
- Call *malloc* or *calloc* to allocate the required memory.
- Cast the returned pointer (of type **void\***, “unknown type of pointer”) to the desired pointer type.
- Store it somewhere so that you can use it later.
- Initialize the memory, and use it.
- When the memory is no longer needed, call *free* to free up the memory. Make sure nowhere else in the program will use the freed pointer.

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. The

library won't care that it is used to store 10 integers. Note that the **casting syntax is simply (type)value**. There is nothing called **static\_cast**, **reinterpret\_cast** or **dynamic\_cast** in C.

Strictly speaking, the casting is not necessary, since C allows pointers of type **void \*** (returned by *malloc* and *calloc*) to automatically be converted (“coerce”) to any type of pointer. But some old libraries returns a **char \*** instead of a **void \***, causing problems. The C++ language also doesn't perform such coercion. So it is better to perform the casting anyway for portability.

## 8. C-style strings

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str = "hello, world!";
    cout << str.substr(7, 5) << endl;
}

#include <string.h>
int main() {
    char* str = "hello, world!\n";
    char* tmpstr = strdup(str+7);
    tmpstr[5] = '\0';
    puts(tmpstr);
    free(tmpstr);
}
```

The C++ library provides the *string* class to represent a sequence of characters, which is a vector of characters with some convenient built-in member functions. In contrast, the C library provides no such type. The **char \*** type is in general used for such purpose. While a C++ *string* contains both the length and the content, a **char \*** contains only the contents. The programmer must use their own ways to represent the length, separated from the **char \***. Most strings guarantee to contain no “null character” (i.e., the character with ASCII 0), and in those strings one can use the null character as the terminator, avoiding the need to separately store a length. So a 5-character string is actually 6 byte long, the last one being the null terminator. These are called **C-style strings**, and a set of standard library functions with names starting with *str* (like *strlen*, *strcpy*, *strdup*, *strchr*, etc) defined in *<string.h>* uses this convention. Another way is to use a separated integer to store the length, which is supported by another set of functions starting with *mem* (like *memcpy*, *memchr*, etc).

Since **char \*** are really pointers, pointer arithmetics and pointer-array equivalence play important role. You can see from above that simply using *str+7* will skip the first 7 characters of a string, and directly putting a null character will strip out the right end of a string. To convert an integer to a string, see Section 10.

## 9. Templates versus macros

```
const double PI=3.141592653589793;
template <class T>
T max(T a, T b) {
    return a>b?a:b;
}

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

In C++, we have a facility called **template** which can be used to make functions automatically. C contains no such facility. Instead, **C uses a simplistic string-based system called macro expansion**. A macro can be function-like or constant-like, as seen above. Note the large number of parentheses used in the macro, to make sure that precedence surprise can never happen (think about what will happen if we don't have parentheses and we ask *max(a & 255, b)*). But there is

one problem: the operands are evaluated twice instead of once. For example, if we call  $max(f(), 2)$ ,  $f()$  is called to see whether it is larger than 2. If it is the case,  $f()$  is called again, possibly resulting in another value. Even if it is the same value, it is slower than using the old value again. For these reasons, macros are not usually used in C++ where template is available.

## 10. Input and Output

```
// With <iostream> included
string name;
int number;
cout << "What is your name? ";
cin >> name;
cout << "Hello, " << name << endl;
cout << "Give me a number: ";
cin >> number;
cout << "I get " << number << endl;

/* With <stdio.h> included */
char name[80];
int number;
printf("What is your name? ");
scanf("%s", name);
printf("Hello, %s\n", name);
printf("Give me a number: ");
scanf("%d", &number);
printf("I get %d\n", number);
```

C uses a very simple system for input and output. It is done with functions like *printf* and *scanf* in *stdio.h* rather than objects like *cout* and *cin* in *iostream*. There is no manipulators, so to print the end-of-line characters we simply print `"\n"`.

Without operator overloading, the C library cannot figure out the type of variables by itself. You must tell the library what type and format to use. It is done by a **conversion specifier** like `"%s"`. **Each *printf* prints one string** (the first argument) that may contain conversion specifiers, each will be modified by a variable that follows. E.g., in the above, the second *printf* prints the string `"Hello, %s\n"`, modified by replacing `"%s"` by the C-style string value of *name*. Other commonly used types include `"%d"` (decimal integer), `"%o"` (octal integer), `"%x"` (hexadecimal integer)<sup>1</sup>, `"%f"` (float in floating point format), `"%e"` (float in exponential format), `"%c"` (characters), `"%lf"` (double in floating point format), `"%p"` (pointers), etc. There are more complicated modifications, e.g., `"%5d"` is a decimal number of at least 5 characters long, and `"%6.2f"` is a floating point number of at least 6 characters long with 2 digits after the decimal point.

Input is just similar: the format string specifies the format of the string to expect, and each conversion specifier correspond to a variable that follows. The variables must be pointer-type, as explained in Section 5. But there is one thing to watch out: usually, C uses exactly the same rule as C++ on white-spaces. Before reading an item (e.g., integer), all spaces will be discarded; and characters are read into the item until an offending character is found, which is left in the stream for reading the next time. **But if the item is a char, no whitespace is discarded.** If you want the library to discard spaces, you **must** add a space character before the format specifier. E.g.,

```
char c;
cout << "Continue? (y/n): ";
cin >> c;

char c;
printf("Continue? (y/n): ");
scanf(" %c", &c); /* a space before % */
```

Just like C++ can use `<<` and `>>` to output to a string or a file, C can use *printf* and *scanf* like functions to output to strings (using *sprintf* and *sscanf*) or files (using *fprintf* and *fscanf*). For example:

---

<sup>1</sup>“Decimal”, “octal” and “hexidecimal” are only used when performing input and output. The computer always use a binary format (usually 2’s complement for signed integers, and mantissa-exponent for floating point numbers) to store the number internally, and typically you will never see them in such format.

```
// With <sstream> and <string> included      /* With <stdio.h> included */
ostringstream oss;                          char s[20];
/* store "10.20" into s */                 double val;
oss << 10 << '.' << 20;                     /* store "10.20" into s */
string s = oss.str();                       sprintf(s, "%d.%d", 10, 20);
/* store 10.2 in val */                    /* store 10.2 in val */
istringstream iss(s);                       sscanf(s, "%lf", &val);
double d;
iss >> d;
```

There are other functions that can also be useful, including *putc*, *puts*, *getc*, *fgets*, etc. **All standard C functions are documented in man pages.** Depending on whether they are “system calls” (we will know what it means soon), they are in either section 2 or 3. Since *printf* and *scanf* are not system calls, their man pages are in section 3. Type, e.g., `man 3 printf` (or `man -s 3 printf` in Solaris) to read documentation for *printf*.

## 11. Function pointers

All good C programmers know **function pointers**<sup>1</sup>, a type of pointer values that point not to a variables but instead to a function. The name of every function is a function pointer literal (constant), just like the name of every array is a pointer literal. For example, if we define

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

Then *my\_sqrt* is a function pointer of type **double(\*)**(double)****. That is, it is a pointer pointing to a function that takes a **double** type argument and return a **double** result<sup>2</sup>. We can create variables or arguments of such function pointer type, or store them in struct fields, just like values of other types. The only use of the function pointer value is to call the corresponding function. For example, the following function receives such a pointer and call it.

```
void callit(double (*p)(double)) {
    p(2.0);      /* or more verbosely (*p)(2.0) */
}
```

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

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

This facility is used in many library functions, including *qsort* and *bsearch* which performs quick sort and binary search. It is also useful for implementing object-oriented programs in C, but it is quite involved and is not covered in this note.

---

<sup>1</sup>In C++ one usually use a function object instead, which has an **operator()** function defined.

<sup>2</sup>Why **double (\*)**(double)**** means such pointer? In general, a C declaration like **double (\*)**(double)**** should be read from inside-out. Here *p* is a pointer, since the next thing that binds to it is the \*. Taking that out we have **double ???**(double)****, and so pointer is to a function of that type. Alternatively, you can say that *p* is a type of variable that will give you **double** if used like *(\*p)(2.0)*. We say that in C, the declaration syntax is the same as the usage syntax.