CSCI 240 Lecture Notes - Part 4

Three important ideas so far:

- program = data + instructions
- data is characterized by its *data type* which defines what operations can be performed with it and what *values* it can hold
- instructions are written as structures: sequence, repetition, decision

Next important idea:

- programs can be broken up into functions, each of which does part of the job.

Functions

As a first introduction, understand that the *main()* function will become the "boss" function which calls on "worker" sub-functions to do pieces of the work.

We will learn to use functions written by others and also how to write and use our own. Sets, or "libraries" of such functions are the main way in which the C++ programming language can be extended or expanded. For example, there are libraries of mathematical functions, libraries of graphical functions, libraries of I/O functions, and many more.

These worker sub-functions are relatively small and simple (compared to the whole program, which does a larger, more complicated task than any one of its sub-functions).

We can make an analogy of a building *contractor*. *main()* who hires subcontractors (plumber, roofer, electrician - the functions) to do parts of the whole job. (The plumber may hire sub sub-contractors to do part of the plumbing job (diggers to get to the sewer, etc.). We will see that in C++ sub-functions can themselves call on other functions to do part of *their* work.

We decompose a program into functions for several reasons:

- to hide irrelevant detail at the *main()* level, so the the program's primary purpose is clearer
- to divide a complex problem into a series of simple problems
- to make subsequent modification of the program easier
- to reduce the errors that inevitably come with a single large complex program

Note on terminology:

When one function uses another, we say that it **calls** the other. For example, if *main()* uses the *sqrt()* function to calculate a square root, we say that *main()* **calls** *sqrt()* - and that *sqrt()* is **called** by *main()*. So the boss is the **caller** function and the worker is the **called** function.

Boss's caller's view of a function

The called function is a black box that does a job:

1. It may need to be given certain information to do its job (called *arguments*). For example, *sqrt()* needs to be given the number whose square root is to be found
2. It then does its job (from caller's point of view, we don't care how)
3. It may "return" a result or answer to the caller.

```
Arg1 =>                          => a result
Arg2 =>
```

Example: a square root function. Takes (i.e. must be given) a real number as an argument and returns its square root:

```
num => [sqrt fn] => sqrt_of_num
```

In *main()* (or the calling program), we code:

```
sqrt_of_num = sqrt(num);
```

- *num* is the (single) argument - the info that *sqrt()* needs to get an answer. It is a numeric expression.
- *sqrt* is the name of the function
- *sqrt_of_num* is the name of a variable (float or double) into which the result of the function call is stored.
- *sqrt(num)* is the function call to the function. In a sense, we could also say that the function call *sqrt(num)* **becomes** a value, and that (in this example) the value is then assigned to *sqrt_of_num*.
- In this case, the result was stored in a variable. However, we use a function result anywhere we can code an expression; e.g.:

```
cout << sqrt(num);              or          y = (b + sqrt(b*b - 4*a*c))/(2*a);
```

Notice that from boss's or caller's point of view, we don't know or care how the function does its work. This leaves the client free to concentrate on overall program logic and structure.

The *sqrt()* function is in a standard library. In some systems, you must

```
#include <math>
```

to make this library accessible to your program.

Function's view of its boss or caller

NONE. Except the arguments passed.

That is, the function has no idea what the boss is doing.

It cannot see or use or access the boss's plan, logic, variables, or anything.

The function just does its job with the information (arguments) given to it and it may (depending on how it is written) return a single result (an answer).

Advantages:

1. divide and conquer strategy.
2. a function can be called from multiple places in a program. This avoids code duplication.
3. a function can be used in other programs easily (in a library. Ex: *sqrt()*)
4. makes maintenance easier, since any change (fix a bug, improve the speed) can be done in one place, not everywhere the code occurs
5. makes code more reliable, less prone to bugs - more later

Disadvantages:

1. a little more up-front planning work
2. sometimes a few more lines of code

How to Write a Function

Decide

1. name
2. what the function needs to return (if anything) including the data type of the returned result.
3. what information it needs to do its job (its arguments)
4. what local or temporary variables it needs to do its job (see below)
5. the logic of the function

Example 1: we want a function to cube an integer. No error checking.

1. Name: *cube*
2. Returns: an integer - the cube of a number.
3. Arguments: one argument - the number to be cubed
4. Local variables: an integer to hold the calculated result
5. Logic: trivial

```
int cube( int x )     //See Note 1
{
int result;

result = x * x * x;

return result;          // See Note 2
}
```

Note 1:

int - tells the data type of the value to be returned
cube - the name of the function
int x - tells the data type of the arg, and gives it a name to be used inside the function. This name may or may not be the name of a variable in the caller. It doesn't matter.

Note 2:

return value; - here you supply the value to be returned. In this case it must be an integer expression since the function has *promised* to return an integer.

In boss or caller you could use the function *cube()* as follows:

```
   int num, ans;

   num = 2;
   ans = cube(num);              // 8
   cout << ans;

   ans = cube(ans);              // 512
   cout << ans;

   ans = cube(3);                // 27
   cout << ans;

   cout << cube(cube(3));        // 729
```

Note that 3 different things were passed to *cube()* in the first three different calls: *num*, *ans*, and *3*.

Inside *cube()*, each passed in value - was called x.

cube() does not know the names of the variables in the caller (*num*, and *ans*; and 3 is not even the name of a variable).

In fact, *cube()* does not even get access to *num*, *ans*, or *3* or *9*. **It gets copies of their values.** It uses the copies to do its work. Since it gets copies, it **cannot change the original data passed**. (Think about this - what would it mean to alter the value of 3???)

Example 2: similar to 1, but we want to return the cube of half the number passed as the argument.

```
int cube_half( int x )
{
x = x / 2;

return x * x * x;
}
```

Here x (the "value" for the argument passed from the caller) has been *changed* in the function, but understand that only the *copy* is changed. The original (back in the calling program) is unchanged.

in caller:

```
   num = 4;
   ans = cube_half(num);

   cout << ans << num;     // displays 8 4
```

num is still 4. It was not altered by the division within *cube_half()*. Only the *copy of num* was halved in *cube_half()*.

Example 3: Sometimes functions don't need to return a value. Most often that's when they just print something.

Then we say they return *void*.

Here's a function to display a page header with page number at the top of the page. (Assume *cout*'s are sending output to the printer here; note'n' = tab and 'f = newpage for the printer)

```
void pageHeader(int pagenum)
{
cout << "\t\tInventory Control System\tpage " << pagenum;
cout << "\nNewprint\n\tQuantity\t\tTitle\n";
}
```

Notes:

1. 'if is formfeed = new page

2. '\n is a tab

3. nothing is returned, so the *return* statement is not required, but it can be added with no arguments. In a *void* function, if there is no return statement, control returns to the calling program after the last line in the function is executed.

In caller, you'd code:

```
pageHeader(1);
```

or, better, keep the current page number in an int variable, initialized to 0 and incremented before printing each page:

```
int pnum = 0;

//these two would probably be in a loop that prints 1 page per repetition

pnum++;
pageHeader(pnum);
```

Example 4a: write a function to display two dashed lines on the screen. Like this:

```
----------
----------
```

The number of dashes is determined by the caller by supplying that number as an argument to the function.

```
void lines(int n)
{
int i;  //NOTE: use of a local variable

cout << '\n';

for (i = 1; i <= n; i++)
  cout << "-";

cout << '\n';

for (i = 1; i <= n; i++)
  cout << "-";
}
```

Notes on i:

- i here is a *local variable*.
- It is used only in the function
- It is invisible outside the function
- It does not retain any value between function calls. In fact, it does not exist between calls to *lines()*. It only exists when *lines()* is executing.
- Other functions (including the calling function) can have their own variable named i
- Typical calling statement would be: *lines(40);*

Example 4b: write a function to display *n* dashed lines each of which has *m* dashes in it. So a calling statement would be, for example,

```
dashes(3, 40);    //make 3 lines of 40 dashes
```

We'll write this as a function *dashes()* which calls a sub-function (in a loop) *oneLine()* to do one line:

```
//display numLines lines of dashes

void dashes(int numLines, int numDashes)
{
int i;

for (i = 1; i <= numLines; i++)
  oneLine(numDashes);
}


//display one line of num dashes on a new line

void oneLine(int num)
{
int i;

cout << '\n';

for (i = 1; i <= num; i++)
  cout << "-";
}
```

Study this example and be sure you understand how it works.

- Notice that each function is quite simple.
- Notice that the local loop counter variable i called i in each function, but since functions know nothing about each other, this is not a problem.
- Notice that the loop that does dashes goes from 0 to <= num instead of 1 to <= num as in Example 4a - but that the same number of dashes are produced, given the same argument
- Notice that the name of the argument to *oneLine()* is not the same as its alias inside *oneLine()*
- etc. etc.

Example 5: Write a function to raise an integer number to an integer power. Assume both arguments are > 0; also assume that the result will fit in a int. Since n to the i could generate a really big value, you would have to be careful that you don't exceed the maximum integer, or perhaps use a special data type with a larger range.

```
int n_tothe_i(int n, int i) //note 2 args passed
{
int ndx;
int temp;

//temp = n to the first power

temp = n;

//now mult temp by n i-1 times

for (ndx = 1; ndx < i; ndx++)
  temp *= n;

return (temp);
}
```

("Play computer" with this to satisfy yourself that it works correctly.)

Sample calling statements:

```
x = n_tothe_i(3, 2);     //3 to the 2 power

y = n_tothe_i(a, 4);

x = n_tothe_i(x, (i*j));  //note use of expr as arg
```

Example 6: Write a function that returns the (double) average of a sum and *count* (passed as a double and an int) **if** the value in *count* is greater than 0. Have it return 0 if *count* is 0 or negative. (Yes, I know that n/0 is not defined and shouldn't be set to 0. And that 0/n is 0. This is just for illustration.)

```
double avg(double sum, int count)
{
if (count <= 0)
  {
  return 0;
  }
else
  {
  return sum/count;
  }
}
```

Note that there are 2 return statements here. The first one executed is the last statement executed in the function. It is an immediate bailout. Some people discourage the use of multiple returns in a function. They argue that in a large function, you may not see all the return statements and so you might misunderstand how the function works. In small simple functions, this is usually not a big problem. However, if you want to have just one return, you could re-write the function as follows. You'll need one local variable:

```
double avg(double sum, int count)
{
double temp;

if (count <= 0)
  {
  temp = 0;
  }
else
  {
  temp = sum/count;
  }

return temp;
}
```

One further requirement for using functions:

You must tell the C++ compiler a bit about each function you use in your program *before* you use it.

After the #includes (and any #defines), but before *int main()*, add one line for each function that you are writing and using in this source code file.

These lines are called the *function prototypes*.

Each line is of the following form:

type *function-name*(arg1 type, arg2 type, ...);

The arguments need only be listed by **type** - you don't have to write a name here. (But you can include a name if you'd like to remind yourself of what the argument stands for.)

If a function takes no arguments, you can just type a set of empty ().

Note the semi-colon at the end of each line. It is **required** for function prototypes.

Examples - prototypes for all the previous functions

```
int cube(int);
int cube_half(int);
int n_tothe_i(int,int);
void lines(int);
void dashes(int, int);
void oneLine(int);
int n_tothe_i(int, int);
double avg(double, int);
```

Finally, how do all these pieces go together? Here's a skeleton program that uses *cube()*

```
#includes...

using statement

#defines...


int cube(int);

//*************************************************************

int main()
{
int ans;

ans = cube(5);
cout << ans;

return 0;
}

//*************************************************************

int cube(int x)
{
int result;

result = x * x * x;

return result;
}
```

Notes:

- Precede every function with a //************ - to make it easier to find them, and to remind you that each function is a separate entity and cannot see anything in any other function.
- Code *main()* first - **and it, including its opening '{' and its closing '}'** - and then any other functions you write, in any order you feel is useful.
- Make the order of the prototype statements match the order of the actual functions in the listing - just to make them easier to find.