An example to illustrate the difference between actual and formal parameters

In this program, the user will enter four values. The program will print out the sum of the pair of largest values.

A function that may help us in this program is one that determines the maximum of three integers. Here is such a function:

```
int Max3(int a, int b, int c) {

   if ((a > b) && (a > c))
       return a;
   else if (b > c)
       return b;
   else
       return c;
}
```

Now, we can use that function as follows in our program:

```
#include <stdio.h>

int Max3(int a, int b, int c);
int main() {

   int a, b, c, d;
   int max1, max2;

   printf("Enter the four values.\n");
   scanf("%d%d%d%d", &a, &b, &c, &d); // Read input.

   max1 = Max3(a+b, a+c, a+d); // Compute max of three pairs
   max2 = Max3(b+c, b+d, c+d); // Compute the max of the
                               // other three pairs.

   if (max1 > max 2)
      printf("The sum of the two largest values is %d\n", max1);
   else
      printf("The sum of the two largest values is %d\n", max2);

  return  0;
}

// Pre-condition: All parameters are integers.
// Post-condition: The largest value of the three is returned.
int Max3(int a, int b, int c) {

   if ((a > b) && (a > c))
       return a;
   else if (b > c)
       return b;
   else
       return c;
}
```

In this previous example, in the function definition, there were three variables, a, b, and c. These are the formal parameters for the function. In main, we called the Max3 function twice.

On the first call, we passed the following actual parameter:

a+b, a+c, and a+d. Note that the a, b, and c represented here are DIFFERENT than the a, b, and c that are formal parameters in the Max3 function.

In general, each function gets to have its own variables. But within a function you can NOT have two variables with the same name. BUT, you CAN have two variables with the same name in two different functions. Thus, when you trace through code, it makes sense to refer to variables with subscripts, like so: a_{main} or a_{Max3}.

Furthermore, it's also important to realize that the Max3 function can be called multiple times, so that for EACH separate function call, the Max3 function has a copy of each of its own variables. (There are two types of variables a function can have: its formal parameters AND its locally defined variables.) If/When you see recursion in COP 3502, this understanding will be very important.

In the second call to Max3, the formal parameters are b+c, b+d, and c+d. Remember that each time a function call is made, the FIRST thing that occurs is that the VALUES of the ACTUAL parameters get COPIED into the corresponding FORMAL parameter. Then, the function is run. While this is occurring, the ONLY valid variables are formal parameters of the function and other local variables of the function. When the function completes, it returns to the function that called it, and that function resumes exactly from the point it left off.

Another Example

```
#include <stdio.h>

#define     CUR_YEAR        2005
#define     CUR_MONTH       9
#define     CUR_DAY         29

int main() {

    char oldname[20], name[20];
    int oldage, age, mon, theday, yr;
    char ans, dummy;

    // Calculate first person’s age.
    printf(“Enter the first persons name.\n”);
    scanf("%s", oldname);
    printf(“Enter the first persons birthdate, mon, day and yr\n.”)
    scanf("%d%d%d", &mon, &theday, &yr);
    oldage = Find_Age(mon, theday, yr);

    printf(“Are there more people in your group?\n”);
    scanf("%c%c", &ans, &dummy);
    while (ans == 'y') {

       // Look at next person’s age.
        printf(“Enter the next persons name.\”);
        scanf("%s", name);
        printf(“Enter his/her birthdate; month, day and year.\n”);
        scanf("%d%d%d", &mon, &theday, &yr);
        age = Find_Age(mon, theday, yr);

        //Adjust oldest person and age, if necessary.
        if (age > oldage) {
            oldage = age;
            strcpy(oldname, name);
        }
        printf(“Are there more people in your group?\n”);
        scanf("%c%c", &ans, &dummy);
    }

    printf("%s is the oldest person at %d years old.\n", oldname,
                                                          oldage);
    return 0;
}

// Precondition: Each parameter is a positive integer
//                      representing the month, day and year of the
//                      user’s birth.
// Postcondition: The person’s age will be returned.
int Find_Age(int month, int day, int year)

    int age;
    age = CUR_YEAR – year;

    // Adjust age based on time of year of the user’s birthday.
    if (month > CUR_MONTH)
        age--;
    else if ((month == CUR_MONTH) && (day > CUR_DAY))
        age--;

    return age;
}
```

Technically speaking, there's a tiny case where this program may output the wrong person. What is that case? How could we redesign this program to deal with that case?

Void functions

Most of the mechanics of void functions are similar to functions that return values. However, there is one KEY difference: a void function does NOT return a value to the calling function. It is quite possible that no value needs to be returned, thus there is no need to do so. However, we CAN still use the return statement in a void function. Here is the syntax:

```
return;
```

If encountered in the middle of a function, this statement ends the function and returns control to where the function was called from.

Since a void function does not return a value, it does not make sense to call it as part of an expression that is being evaluated. Rather, a void function should almost always be called on a line by instead.

An example of a void function you have seen is srand. It takes care of a task and then finishes. Thus, you simply call this function on its own line.

Another simple example of a void function is one that prints out a menu:

```
void menu() {

    printf("Please choose one of the following.\n");
    //....
}
```

You can call this function as follows: menu();

Another situation where void functions might be useful is in a menu driven program where the menu choices are completely unrelated. Here is a skeleton of the main program of such a function:

```
int main() {

    int choice;
    menu();
    scanf("%d", &choice);

    while (choice != 4) {

        if (choice == 1)
            function1();
        else if (choice == 2)
            function2();
        else if (choice == 3)
            function3();
        else if (choice != 4)
            printf("Sorry, please enter your choice again.\n");

        menu();
        scanf("%d", &choice);
    }
}
```

Each of the functions listed would independently perform their own task. They act as mini-mains in some sense. Really, they are just functions that execute, once they are done, there is no need to save any of the local variables that were created while they were run, nor is there any need to communicate any information back to main.

Example using void functions

Printing out patterns with stars was an example we looked at previously. A natural question to ask is if functions could help us out somehow in these programs.

If we take a look at the types of patterns that could be printed out, we see that one type of function that could be useful is a function that prints out a designated character a certain number of times.

In specifying this task, we have two unknowns: The character to be printed, and how many times it will be printed.

Thus, it makes sense for our function to take in two parameters. Furthermore, one parameter should be a character, the character to be printed, and the second an integer specifying the number of times to print the character.

Using this information, we can come up with a function prototype:

```
// Precondition: ch is a printable character and numtimes is
//               a nonnegative integer.
// Postcondition: ch will be printed numtimes times in a row.
//                no newline character will be printed after
//                this.
void printChars(char ch, int numtimes);
```

Now, let's take a look at how we can accomplish this task:

```
void printChars(char ch, int numtimes) {
    int index;
    for (index=0; index<numtimes; index++)
        printf("%c",ch);
}
```

Now, consider using this function to print a triangle of the following shape (in this example n=4):

```
****
 ***
  **
   *
```

```
int main(void) {

    int index, n;
    int numspaces, numstars;

    printf("Enter the size of your triangle.\n");
    scanf("%d", &n);

    for (index=0; index<n; index++) {

        // Determine the number of spaces and stars.
        numspaces = index;
        numstars = n - index;

        // Print spaces first, then stars.
        printChars(' ', numspaces);
        printChars('*', numstars);
        printf("\n");
    }
    return 0;
}
```

This question tests whether or not you understand the mechanics of functions. Determine the output of the following program:

```
#include <stdio.h>
int f(int a, int b, int c);

int main() {

  int a = 2, b = 3, c = 1;

  c = f(a+b, a+c, b+c);
  printf("a=%d b=%d c=%d\n", a, b, c);

  b = f(a, b, c);
  printf("a=%d b=%d c=%d\n", a, b, c);

  a = f( a, b, f(c, b, a) );
  printf("a=%d b=%d c=%d\n", a, b, c);

  system("PAUSE");
  return 0;
}

int f(int a, int b, int c) {

  int sum;
  sum = a + b + c;
  if (sum < a*b)
    return a + b;
  if (sum <= 2*a*b)
    return b + c;

  return a + c;
}
```