Parameter Passing in C and C++
P. Conrad for CS16, UC Santa Barbara, Spring 2010

Some terminology: *caller* and *callee*

PLEASE ALSO READ:
In Etter, 6.4
(Printer friendly pdf of this handout: paramPassing.pdf)

In the"Pointer fun with Binky" video, Nick Parlante introduced the terminology of *pointee* for the "thing a pointer points to". In the same way, it may be helpful to introduce two similar words when talking about a specific function call:

- caller—this is sometimes called the "calling function". It is the function definition in which the function call appears.
- callee—this is the function that is being called.

Examples of caller and callee

In `example1.c` at the left below, there are exactly two function calls that appear in the code as we read from top to bottom. In this case, that also happens to be the order in which the function calls would occur in time.

1. In the first function call, `main` is the caller, and `squared` is the callee.
2. In the second function call, `main` is the caller, and `printf` is the callee (keep in mind that `printf` is a function call).

In `example2.c` at the right below, there are five function calls—I list them here in the order in which they can be found in the code reading from top to bottom. However, this is NOT the order in which they would actually occur in time. In fact, on any given run of the program, depending on whether `argc==2` or `argc!=2`, which functions get called is different.

1. In the first call, `doStuff` is the caller, and `twiceTheValue` is the callee.
2. In the second call, `doStuff` is the caller, and `printf` is the callee.
3. In the third call, `main` is the caller, and `printf` is the callee (This only happens if `argc!=2.`)
4. The fourth and fifth calls are nested: a call to `atoi` is nested inside the actual parameter of the call to `doStuff`
   - for `atoi`, `main` is the caller, and `atoi` is the callee
   - for `doStuff`, `main` is the caller, and `doStuff` is the callee

This should come as no surprise to you at this stage in your study of programming but it is worth mentioning: in general, the order of functions that gets called, and even which functions get called (or not) may depend on the input to the program, and may be different from the order in which the function calls appears when reading top to bottom in the code. Can you work out which functions get called, and in what order, whem `argc==2`, and the case where `argc!=2`?

```
// example1.c from H23, CS16, S10
#include <stdio.h>

int squared(int x)
{
  return x * x;
}

int main()
{
  int a,b;
  a=5;
  b=squared(a);
  printf("a=%i b=%i\n",a,b);
  return 0;
}
```

```
// example2.c from H23, CS16, S10
#include <stdio.h>

int twiceTheValue(int x)
{
  return 2 * x;
}

void doStuff(int a)
{
  int b;
  b=twiceTheValue(a);
  printf("a=%i b=%i\n",a,b);
}

int main(int argc, char *argv[])
{
  if (argc!=2)
    {
      printf("Usage: %s integer\n",argv[0]);
      return 1;
    }
  doStuff(atoi(argv[1]));
  return 0;
}
```

Please turn over for more...

...continued from other side

Two (well, maybe three) ways to pass parameters in C

When passing parameters to a function, there are two ways it can happen—well, three if you count passing an array as a separate way:

1. Passing by value—e.g. `int myFunc(int x);`—the value is copied from an expression in the caller to an expression in the callee
2. Passing by pointer—e.g. `int myFunc(int *p);` a pointer is passed from the caller to the callee.
   - This pointer is often the address of a variable in the caller.
   - This allows the callee to directly access the variable in the caller by dereferencing the pointer.
   - That means the value can be pulled out of the variable in the caller (used on the right hand side of an assignment), e.g. int x = (*p);
   - This means the value of the variable in the caller can also be changed, by dereferencing the pointer on the left hand side of an assignment statement (e.g. (*p) = 5;
3. Passing an array—e.g. `int myFunc(int nums[], int n);` which is really just a special case of passing a pointer—essentially another way of writing `int myFunc(int *nums, int n);`
   - In C and C++, the name of an array is a pointer to the first element of the array.
   - So, when you pass an array name, you are really passing a pointer.
   - And, saying a[i] is really just a shorthand for *(a+i), so when you access an array parameter in C, you are accessing the array in the caller.

We've been over these in lecture, and they are covered in your Etter textbook—so the information above is really just review.

The overworked ampersand

In C++, there is yet one more way to pass parameters, and the syntax can be a bit confusing—it adds yet another meaning for the overworked ampersand. So before we go there, let's review the meaning that the ampersand already has:

1. As a **unary operator in an expression** it means **address of** (e.g. `scanf("%i",&x);`)
   - e.g. the right hand side of an assignment, or an actual parameter in a function call)
2. As a **binary operator** it means **bitwise and** (e.g. `printf("%i\n",x & y);`)
   - see homework H22 and the reading assignment that went with it for a review of *bitwise and*
3. As a **binary operator doubled up** it means *logical and* (e.g. `if (x!=0 && y!=0)`
   - this always returns either 1 or 0, the C values for true or false

C++ adds yet one more way to pass parameters: reference parameters (aliases)

In C++, there is yet one more way to pass parameters. It is very similar to the way passing by pointer works in C, but the syntax is simplified.

In C, to pass by pointer, in the callee, we write something like what you see below in squareIt.c at the left, if, for example, the effect we want is that the variable passed in gets its value squared. Note the following:

- In the callee, the formal parameter is of type int *, that is, pointer to integer
- In the caller, the actual parameter is also of type int *—we have to use the & operator to turn the int into an int *, by taking its address. (As we've discussed, an address is a pointer, and a pointer is an address.)

In C++, we can still write exactly the same code—the example on the left is valid C++. However, there is another option, as shown on the right in the file `squareIt.cpp` (as you may recall, `.cpp` is one of the valid filename endings for C++ code.)

The effect of the C++ code is exactly the same as that of the C code, but the syntax is simplified.

Look at the syntax in the C++ example for the function header: `void squareIt(int &x)` and note that the & here does NOT MEAN ADDRESS OF. This is different from the address of syntax because the & occurs in front of a variable that is a formal parameter—when it occurs here, in the header of a function definition or a function prototype, it means that the parameter is a **reference parameter.** A reference parameter can be thought of "similar passing by pointer, but with automatic transmission as opposed to standard transmission". That is:

- In the function call, you don't have to use the & in front of the actual parameter
- In the function definition body, you don't have to use the * in front of the formal parameter variable to dereference it every time it occurs.

A reference parameter creates an **alias**—in this example, inside the function squareIt, x is an alias for a in the main program. (We could also say that in the C example, (*p) is an alias for a.)

```
// squareIt.c            // squareIt.cpp (using C style I/O)
#include <stdio.h>       #include <stdio.h>

void squareIt(int *p)    void squareIt(int &x)
{                        {
  (*p) = (*p) * (*p);      x = x * x;
}                        }

int main()               int main()
{                        {
  int a=5;                 int a=5;
  squareIt(&a);            squareIt(a);
  printf("a=%i\n",a);      printf("a=%i\n",a);
  return 0;                return 0;
}                        }
```