# C Review

## Functions

| K&R C (old) | ANSI C |
|---|---|
| Declaration | Prototype |
| int fcomp(); | int fcomp(int a, int b);<br>or<br>int fcomp(int, int); ← preferable |
| Definition | Definition |
| int fcomp(a,b)<br>int a;<br>int b;<br>{<br>…<br>} | int fcomp(int a, int b){<br>…<br>} |

The use of a function requires a prototype. For functions like printf(), the function prototypes are contained in stdio.h or related headers.

Not so much an issue now, but problems can occur if one mixes K&R C with ANSI C in function declarations and definitions in separate files. This is because under K&R C, type promotion can occur because the compiler makes everything a standard size, int, double, pointer (and thus types might have changed from what you expected)

# C Review

```c
#include <stdio.h>
```

Notice parenthesis around the variable t and body of macro

```c
#define  abs(t)       (((t)>=0)?(t):-(t))
#define  square(t)   ((t)*(t))
```

No semicolon

```c
double sqr(double);
```

Square Root Function Prototype

```c
void main(void) {

    int i;
    double c;

    for( i=1; i<11; i++){
        c = sqr(i);
        printf(" \t  %d  \t  %f  \t  %f  \n", i, c, square(c));
    }

}
```

```c
double sqr( double x ){
    double x1=1;
    double x2=1;
    double c;
    do{
        x1=x2;
        x2=(x1 + x/x1)/2;
        c = x1-x2;
    }while( abs(c) >= 0.00000001);
    return x2;
}
```

Why not:
while( abs(x1-x2) >=…)  ?

Because the macro would cause the argument x1-x2 to be evaluated three times. (speed issue)

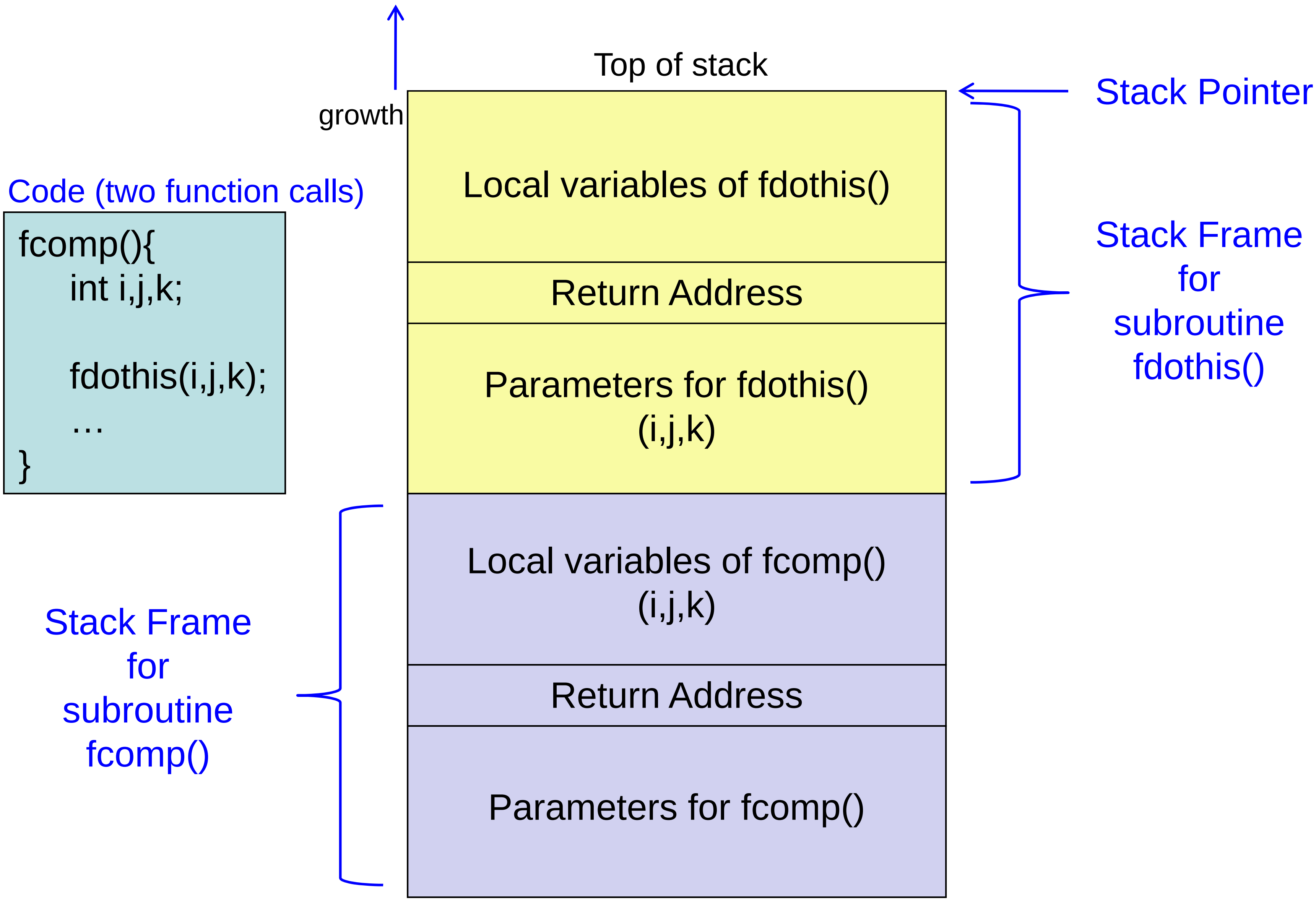
## C Review

### The function call stack

Code (two function calls)

```
fcomp(){
    int i,j,k;

    fdothis(i,j,k);
    …
}
```



## C Review

C requires a function to have an argument list
even if there are no arguments

If f is a function;

```
f();
```

is a statement that calls the function, but

```
f;
```

does nothing at all.

More precisely, it evaluates the address of the function
but does not call it.

# C Review

## Recursion

C permits a function to call itself

Example: factorial

n! = n * (n-1) * (n-2) * … * 2 * 1;

n! = n * (n-1)!

```
long fact( int n ) {
    if (n == 0){
        return 1;
    }else{
        return n*fact(n-1);
    }
}
```

**Hardware/Software Tradeoff**
Recursion is not used much in embedded systems because of limited stack space.
Recursion can cause stack overflow problems.

# C Review

## Variable number of arguments

Suppose you wanted to create a function that could take an arbitrary number of parameters.

```
double ave(int N, double a1, double a2,  …. double aN)
```

```
double ave(int N, …) // This is how your function would be written in C
{
…
}
```

You will need to use the following include file and functions

```
# include <stdarg.h>  // necessary header file

void va_start(va_list  ap,  parmN);  // parmN is the last fixed parameter

type va_arg(va_list  ap,  type);  // returns the next type value

void va_end(va_list  ap);  // called when all the arguments have been processed.
```

# C Review

## Variable argument example

```
#include <stdarg.h>

double average(int count, ...) {
    va_list ap;
    int j;
    double tot = 0;

    va_start(ap, count); //Requires the last fixed parameter (to get the address of it)

    for(j=0; j<count; j++) {
        tot+=va_arg(ap, double);  //Requires the type for type casting.
     }                            // Increments ap to the next argument.
    va_end(ap);

    return tot/count;
}
```

# C Review

## Pointers

A pointer to a variable is the ***address*** of a variable

```
int *px;
int x;

px = &x;  // the ampersand & notifies the compiler to use the
          // address of x rather than the value of x.

x  =  *px; // * is a unary operator that applies to a pointer and
           // directs the compiler to use the integer pointed to by
           // the pointer px.

           // * is referred to as the dereference operator

           // Remember: if px is a pointer to an int, *px is the int.
```

# C Review

## Pointers

A pointer to a variable is the ***address*** of a variable

```
int *px;    // Note:  this provides memory space for the pointer to the
            // type int, but does not provide any memory for the
            // int itself.
int x;      // allocates memory for the int itself

px = &x;    // the ampersand & notifies the compiler to use the
            // address of x rather than the value of x.

x  =  *px;  // * is a unary operator that applies to a pointer and
            // directs the compiler to use the integer pointed to by
            // the pointer px.

            // * is referred to as the dereference operator

            // Remember: if px is a pointer to an int, *px is an int.
```

## C Review

Pointers

What does n = ?

```
int m,n;
int *pm;

m   = 10;
pm = &m;
n    = *pm;
```

n = 10

## C Review

Unary pointer operators (*) have higher precedence than arithmetic operators;

What do the following statements do?

```
int *pi;

*pi = *pi + 10;
```

// The integer pointed to by pi will increase by 10.

```
y = *pi + 1;
```

// y will be replaced by one more than the integer
// pointed to by the pointer pi

```
*pi += 1;
```

// the integer pointed to by pi will increase by 1

# C Review

Unary pointer operators (*,++) have the same precedence,
but are associated right to left.

What do the following statements do?

```
int *pi;

++*pi;
```
// The integer pointed to by pi is increased by 1

```
*pi++;
```
// The pointer pi is incremented *after* the
// dereference operator is applied.

```
(*pi)++;
```
// The integer pointed to by pi is post-incremented.

```
*++pi;
```
// The pointer is pre-incremented before the
// dereference.

## C Review

### Pointers and Arrays

```
int *pa;
int a[20];

pa = a;
pa = &a[0];      // The pointer points to the beginning address;

pa++             // The pointer increments to the next location in the array.
pa = pa + 1;     // If pa points to an integer, p++ points to the next integer
pa += 1;         // If pa points to a long,     p++ points to the next long
                 // If pa points to a double,   p++ points to the next double

                 // C automatically takes care of knowing the number of bytes
                 // that must be added to a pointer to point to the next element
                 // in an array.

pa = &a[0];      // *pa       = first element
                 // *(pa+1) = second element
                 // *(pa+2) = third element, etc.
```

## C Review

### Pointer Operations

There is a set of arithmetic operations that can be applied to pointers. They have to be of like types and pointing within the same array.

- Pointers can be compared.
- Pointers can be subtracted. (The result will be the number of elements between the two pointers, not the difference in values of the pointers.)
- Pointers can be incremented or decremented. (The result will be a pointer that points to the next (previous) element.)
- Pointers can be assigned. (It is a good practice to initially assign pointers to NULL before use. Why?)