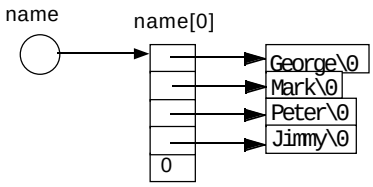
An Array of Strings

char* name[ ] = {"George", "Mark", "Peter", "Jimmy", NULL};

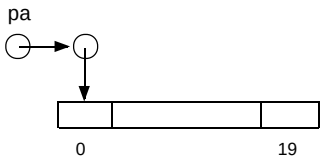


To check syntax of C declarations use:

cmput> ~tony/Cdecl/cdecl

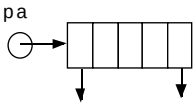
cdecl> explain int (*pa)[20];

declare pa as a pointer to array 20 of int



cdecl> explain int* pa[5];

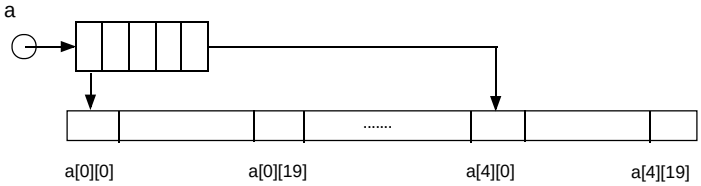
declare pa as array 5 of pointer to int



See also King P. 411 for a simple method for decoding declarations. This information will be extremely useful in a week or two when we discuss pointer functions and pointers to functions.

cdecl> explain int a[5][20];

declare a as array 5 of array 20 of int



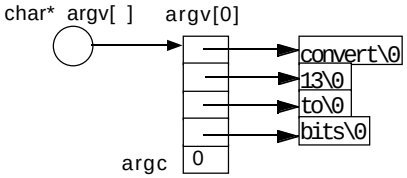
Unix Command line arguments

Consider the prototype:

int main (int argc, char* argv[ ]);

In the above the declaration of argv will produce an access array that guarantees that argv[argc] is NULL

For the command line> convert 13 to bits




```
int main (int argc, char* argv[ ]) {
   int i;
    for (i = 0; i <= argc; i++) {
     if (argv[i] == NULL)
          printf ("Null pointer at: %d\n", i);
     else
          printf ("%s\n", argv[i]);
   }
    return 0;
}
```


Functions and Procedures

- There are two ways of declaring and defining procedures in C, the old way and the ANSI standard way - you may run into both, but use only ANSI form
- All procedures in C are really functions, that is, they return a value. The special type **void** indicates that a return value is not used (not wanted). Such void functions are called procedures.
- C has both function declarations and function definitions - these are two different concepts
- A **function declaration** contains all the information required to call the function, that is the name the type of each parameter and the type of the return value. These declarations are also called **prototypes**.
- A **function definition** includes all the information in a function declaration, plus local variables and the statements in the function - It not only describes how the function can be called, but also how it computes its value

Function Declarations (ANSI prototype)

type FunctionName (ParameterDeclarations);

- For example:
  float Work (int x, double y, char* s);
  float Work (int , double , char*);
- The parameters in the prototype are in the order of the function definition (no surprise there).

Function Definitions

- The ANSI style of function definition is:

```
type FunctionName (ParameterDeclarations) {
     <local variable declarations>;
     <statements>;
     return (<expression>);
}
```

- A return statement is used to supply the value of the function and to give control back to the calling program
- Formats of the return statement are:

return (Rvalue); or return Rvalue;

and

return ; /* used only with **void** functions */

Although the return can be omitted in that case.

Parameter Passing

- All parameter passing in C is by value, that is, when a function is called the parameter values from the calling function are copied into temporary storage in the called function--all modifications to the parameter values occur in this temporary storage; the original values in the calling function are not changed

- This means that you cannot return a result directly through a parameter. The only way to export a result with a parameter is to do so indirectly through a pointer to the location where the result will be stored

- Remember arrays are built with pointers, so if you pass an array address (not an array element) to a function, you can modify the elements in the array and these changes will be seen outside of the function

```
void initialize (int x) {

    we can do anything we like to x inside this function.
    The calling function won't see any of these changes.
    It provides only the initial value of x
    x = 5;
}
```

but consider ::::::::


```
void reset (int* y) {

    *y = 5;          /* put 5 in cell pointed to by y */
}

    int i = 3, j = 3;
    initialize ( i );
    reset ( &j );
    printf ( "%d, %d", i, j );
```

- This will print 3, 5, since we have passed a pointer to j into the function, the location pointed to is changed by the reset - note that the invocation **reset (j)** will cause all sorts of problems, since the parameter **j** isn't a pointer.

When would j be a pointer?

```
    int    j[10];
```

then

```
    reset (j);
```

is the same as `reset ( &j[0] );`

thus the parameter j is a pointer. HOWEVER, the corresponding formal parameter would still be

```
    void reset ( int* y );
```


The ReadLine and FetchLine functions, below, are equivalent.

```
#include <stdio.h>
int ReadLine ( char str[ ], int n )
{
    int ch;
    int i = 0;

    while ( (ch = getchar()) != '\n' ) {
        if ( i < n ) {
            str[i] = ch; i++;
        }
    }
    str[i] = '\0';
    return (i);
}

int FetchLine (char* str, int n)
{
    int ch;
    int i = 0;

    while ( (ch = getchar()) != '\n' ) {
        if ( i < n ) {
            *(str+i) = ch; i++;
        }
    }
    *(str+i) = '\0';
    return (i);
}
```

The parameter declarations are consistent with usage.


```
void main (void) {

    char message [32];
    int m;

    m = ReadLine (message, 16);
    printf ("%d chars in: %s\n", m, message);
    m = FetchLine (&message[0], 16);
    printf ("%d chars in: %s\n", m, message);
}
```

Function Calls

When you call a function, the actual parameters (arguments) to the function are copied and placed onto the stack. Thus the function manipulates copies of the parameters, not the original arguments themselves. That is to say, all function calls in C are call by value.

To alter a data object in the calling program you must either return a value from the function, or pass the address of the object to the function.

Just to convince you that arguments are copied:

```
#include <stdio.h>

void look (int a, int b, char c, char d, double f, double g) {
    printf ("The address of argument a is %x\n", (int) &a);
    printf ("The address of argument b is %x\n", (int) &b);
    printf ("The address of argument c is %x\n", (int) &c);
    printf ("The address of argument d is %x\n", (int) &d);
    printf ("The address of argument f is %x\n", (int) &f);
    printf ("The address of argument g is %x\n", (int) &g);
}
```

```
/ *
     The previous is a procedure called "look" that prints 8 lines of
     output.  Each line has a single number that is the address of a
     different parameter--coerced from an unsigned int to an int and
     printed in hexadecimal (%x)
* /

int main (void) {
     static int a = 1;              /* 4 bytes of space */
     static int b = 2;
     static char c = '3';           /* 1 byte of space */
     static char d = '4';
     static double f = 5.0;         /* 8 bytes of space */
     static double g = 6.0;

     printf ("The address of a is %x\n", (int) &a);
     printf ("The address of b is %x\n", (int) &b);
     printf ("The address of c is %x\n", (int) &c);
     printf ("The address of d is %x\n", (int) &d);
     printf ("The address of f is %x\n", (int) &f);
     printf ("The address of g is %x\n", (int) &g);

     look (a, b, c, d, f, g);
     return  (0);
}
```

produces output like

```
The address of a is 40b0        4 bytes
The address of b is 40b4        4 bytes
The address of c is 40b8        1 byte
The address of d is 40b9        1 byte + 6 bytes
The address of f is 40c0        why not 40ba ?
The address of g is 40c8        8 bytes
```


Run this program and discuss the remaining output with your TA

```
The address of argument a is effff7ac      4 bytes
The address of argument b is effff7b0      4 – 4 bytes
The address of argument c is effff75f      1 +80 bytes
The address of argument d is effff75e      1 byte
The address of argument f is effff750      8 + 6 bytes
The address of argument g is effff748      8 bytes
```

Note the size of the (stack) addresses!!
Why are they not at 4-byte intervals?