WIKIBOOKS

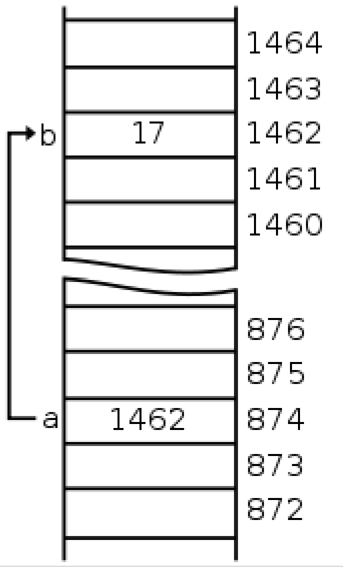
C Programming/Pointers and arrays

A **pointer** is a value that designates the address (i.e., the location in memory), of some value. Pointers are variables that hold a memory location.

There are four fundamental things you need to know about pointers:

- How to declare them (with the address operator '&': `int *pointer = &variable;`)
- How to assign to them (`pointer = NULL;`)
- How to reference the value to which the pointer points (known as *dereferencing*, by using the dereferencing operator '*': `value = *pointer;`)
- How they relate to arrays (the vast majority of arrays in C are simple lists, also called "1 dimensional arrays", but we will briefly cover multi-dimensional arrays with some pointers in a later chapter).

Pointers can reference any data type, even functions. We'll also discuss the relationship of pointers with text strings and the more advanced concept of function pointers.



Pointer *a* pointing to variable *b*. Note that *b* stores a number, whereas *a* stores the address of *b* in memory (1462)

Contents

Declaring pointers

Consider the following snippet of code which declares two pointers:

```
 1  struct MyStruct {
 2      int   m_aNumber;
 3      float num2;
 4  };
 5
 6  int main()
 7  {
 8       int *pJ2;
 9       struct MyStruct *pAnItem;
10  }
```

Lines 1-4 define a structure. Line 8 declares a variable that points to an `int`, and line 9 declares a variable that points to something with structure MyStruct. So to declare a variable as something that points to some type, rather than contains some type, the asterisk (`*`) is placed before the variable name.

In the following, line 1 declares `var1` as a pointer to a long and `var2` as a long and not a pointer to a long. In line 2, `p3` is declared as a pointer to a pointer to an int.

```
long  *  var1, var2;
int   ** p3;
```

Pointer types are often used as parameters to function calls. The following shows how to declare a function which uses a pointer as an argument. Since C passes function arguments by value, in order to allow a function to modify a value from the calling routine, a pointer to the value must be passed. Pointers to structures are also used as function arguments even when nothing in the struct will be modified in the function. This is done to avoid copying the complete contents of the structure onto the stack. More about pointers as function arguments later.

```
int MyFunction(struct MyStruct *pStruct);
```

Assigning values to pointers

So far we've discussed how to declare pointers. The process of assigning values to pointers is next. To assign the address of a variable to a pointer, the & or 'address of' operator is used.

```
int myInt;
int *pPointer;
struct MyStruct dvorak;
struct MyStruct *pKeyboard;

pPointer = &myInt;
pKeyboard = &dvorak;
```

Here, pPointer will now reference myInt and pKeyboard will reference dvorak.

Pointers can also be assigned to reference dynamically allocated memory. The malloc() and calloc() functions are often used to do this.

```
#include <stdlib.h>
/* ... */
struct MyStruct *pKeyboard;
/* ... */
pKeyboard = malloc(sizeof *pKeyboard);
```

The malloc function returns a pointer to dynamically allocated memory (or NULL if unsuccessful). The size of this memory will be appropriately sized to contain the MyStruct structure.

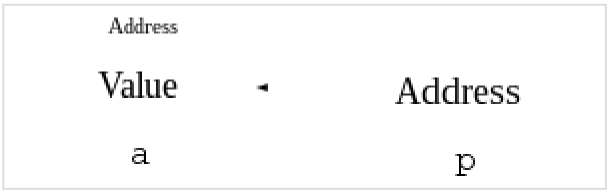
The following is an example showing one pointer being assigned to another and of a pointer being assigned a return value from a function.

```
static struct MyStruct val1, val2, val3, val4;

struct MyStruct *ASillyFunction( int b )
{
    struct MyStruct *myReturn;

    if (b == 1) myReturn = &val1;
    else if (b==2) myReturn = &val2;
    else if (b==3) myReturn = &val3;
    else myReturn = &val4;

    return myReturn;
}

struct MyStruct *strPointer;
int     *c, *d;
int     j;

c = &j;                           /* pointer assigned using & operator */
d = c;                            /* assign one pointer to another     */
strPointer = ASillyFunction( 3 ); /* pointer returned from a function. */
```

When returning a pointer from a function, do not return a pointer that points to a value that is local to the function or that is a pointer to a function argument. Pointers to local variables become invalid when the function exits. In the above function, the value returned points to a static variable. Returning a pointer to dynamically allocated memory is also valid.

Pointer dereferencing

To access a value to which a pointer points, the * operator is used. Another operator, the -> operator is used in conjunction with pointers to structures. Here's a short example.



The pointer p points to the variable a.

```
int   c, d;
int   *pj;
struct MyStruct astruct;
struct MyStruct *bb;

c   = 10;
pj  = &c;             /* pj points to c */
d   = *pj;            /* d is assigned the value to which pj points, 10
*/
```

```
pj  = &d;              /* now points to d */
*pj = 12;              /* d is now 12 */

bb = &astruct;
(*bb).m_aNumber = 3;  /* assigns 3 to the m_aNumber member of astruct */
bb->num2 = 44.3;      /* assigns 44.3 to the num2 member of astruct   */
*pj = bb->m_aNumber;  /* equivalent to d = astruct.m_aNumber;          */
```

The expression `bb->m_aNumber` is entirely equivalent to `(*bb).m_aNumber`. They both access the `m_aNumber` element of the structure pointed to by `bb`. There is one more way of dereferencing a pointer, which will be discussed in the following section.

When dereferencing a pointer that points to an invalid memory location, an error often occurs which results in the program terminating. The error is often reported as a segmentation error. A common cause of this is failure to initialize a pointer before trying to dereference it.

C is known for giving you just enough rope to hang yourself, and pointer dereferencing is a prime example. You are quite free to write code that accesses memory outside that which you have explicitly requested from the system. And many times, that memory may appear as available to your program due to the vagaries of system memory allocation. However, even if 99 executions allow your program to run without fault, that 100th execution may be the time when your "memory pilfering" is caught by the system and the program fails. Be careful to ensure that your pointer offsets are within the bounds of allocated memory!

The declaration `void *somePointer;` is used to declare a pointer of some nonspecified type. You can assign a value to a void pointer, but you must cast the variable to point to some specified type before you can dereference it. Pointer arithmetic is also not valid with `void *` pointers.

Pointers and Arrays

Up to now, we've carefully been avoiding discussing arrays in the context of pointers. The interaction of pointers and arrays can be confusing but here are two fundamental statements about it:

- A variable declared as an array of some type acts as a pointer to that type. When used by itself, it points to the first element of the array.
- A pointer can be indexed like an array name.

The first case often is seen to occur when an array is passed as an argument to a function. The function declares the parameter as a pointer, but the actual argument may be the name of an array. The second case often occurs when accessing dynamically allocated memory.

Let's look at examples of each. In the following code, the call to `calloc()` effectively allocates an array of struct MyStruct items.

```
struct MyStruct {
    int someNumber;
    float otherNumber;
};

float returnSameIfAnyEquals(struct MyStruct *workingArray, int size, int bb)
{
    /* Go through the array and check if any value in someNumber is equal to bb. If
     * any value is, return the value in otherNumber. If no values are equal to bb,
     * return 0.0f. */
    for (int i = 0; i < size; i++) {
        if (workingArray[i].someNumber == bb ) {
            return workingArray[i].otherNumber;
        }
    }
    return 0.0f;
}

// Declare our variables
float  someResult;
int    someSize;
struct MyStruct myArray[4];
struct MyStruct *secondArray; // Notice that this is a pointer
```

```
const int ArraySize = sizeof(myArray) / sizeof(*myArray);

// Initialization of myArray occurs
someResult = returnSameIfAnyEquals(myArray, ArraySize, 4);

secondArray = calloc(someSize, sizeof(struct MyStruct));
for (int i = 0; i < someSize; i++) {
    /* Fill secondArray with some data */
    secondArray[i].someNumber = i * 2;
    secondArray[i].otherNumber = 0.304f * i * i;
}
```

Pointers and array names can pretty much be used interchangeably; however, there are exceptions. You cannot assign a new pointer value to an array name. The array name will always point to the first element of the array. In the function `returnSameIfAnyEquals`, you could however assign a new value to workingArray, as it is just a pointer to the first element of workingArray. It is also valid for a function to return a pointer to one of the array elements from an array passed as an argument to a function. A function should never return a pointer to a local variable, even though the compiler will probably not complain.

When declaring parameters to functions, declaring an array variable without a size is equivalent to declaring a pointer. Often this is done to emphasize the fact that the pointer variable will be used in a manner equivalent to an array.

```
/* Two equivalent function prototypes */

int LittleFunction(int *paramN);
int LittleFunction(int paramN[]);
```

Now we're ready to discuss pointer arithmetic. You can add and subtract integer values to/from pointers. If myArray is declared to be some type of array, the expression `*(myArray+j)`, where `j` is an integer, is equivalent to `myArray[j]`. For instance, in the above example where we had the expression `secondArray[i].otherNumber`, we could have written that as `(*(secondArray+i)).otherNumber` or more simply `(secondArray+i)->otherNumber`.

Note that for addition and subtraction of integers and pointers, the value of the pointer is not adjusted by the integer amount, but is adjusted by the amount multiplied by the size of the type to which the pointer refers in bytes. (For example, `pointer + x` can be thought of as `pointer + (x * sizeof(*type))`.)

One pointer may also be subtracted from another, provided they point to elements of the same array (or the position just beyond the end of the array). If you have a pointer that points to an element of an array, the index of the element is the result when the array name is subtracted from the pointer. Here's an example.

```
struct MyStruct someArray[20];
struct MyStruct *p2;
int i;

/* array initialization .. */

for (p2 = someArray; p2 < someArray+20;  ++p2) {
    if (p2->num2 > testValue)
        break;
}
i = p2 - someArray;
```

You may be wondering how pointers and multidimensional arrays interact. Let's look at this a bit in detail. Suppose A is declared as a two dimensional array of floats (`float A[D1][D2];`) and that pf is declared a pointer to a float. If pf is initialized to point to A[0][0], then *(pf+1) is equivalent to A[0][1] and *(pf+D2) is equivalent to A[1][0]. The elements of the array are stored in row-major order.

```


float A[6][8];
float *pf;
pf = &A[0][0];
```

```
*(pf+1) = 1.3;   /* assigns 1.3 to A[0][1] */
*(pf+8) = 2.3;   /* assigns 2.3 to A[1][0] */
```

Let's look at a slightly different problem. We want to have a two dimensional array, but we don't need to have all the rows the same length. What we do is declare an array of pointers. The second line below declares A as an array of pointers. Each pointer points to a float. Here's some applicable code:

```
float  linearA[30];
float *A[6];

A[0] = linearA;              /*  5 - 0 = 5 elements in row  */
A[1] = linearA + 5;          /* 11 - 5 = 6 elements in row  */
A[2] = linearA + 11;         /* 15 - 11 = 4 elements in row */
A[3] = linearA + 15;         /* 21 - 15 = 6 elements        */
A[4] = linearA + 21;         /* 25 - 21 = 4 elements        */
A[5] = linearA + 25;         /* 30 - 25 = 5 elements        */

*A[3][2] = 3.66;          /* assigns 3.66 to linearA[17];     */
*A[3][-3] = 1.44;         /* refers to linearA[12];
                             negative indices are sometimes useful. But avoid using them as much as possible. */
```

We also note here something curious about array indexing. Suppose `myArray` is an array and `i` is an integer value. The expression `myArray[i]` is equivalent to `i[myArray]`. The first is equivalent to `*(myArray+i)`, and the second is equivalent to `*(i+myArray)`. These turn out to be the same, since the addition is commutative.

Pointers can be used with pre-increment or post-decrement, which is sometimes done within a loop, as in the following example. The increment and decrement applies to the pointer, not to the object to which the pointer refers. In other words, `*pArray++` is equivalent to `*(pArray++)`.

```
long  myArray[20];
long  *pArray;
int   i;

/* Assign values to the entries of myArray */
pArray = myArray;
for (i=0; i<10; ++i) {
    *pArray++ = 5 + 3*i + 12*i*i;
    *pArray++ = 6 + 2*i + 7*i*i;
}
```

Pointers in Function Arguments

Often we need to invoke a function with an argument that is itself a pointer. In many instances, the variable is itself a parameter for the current function and may be a pointer to some type of structure. The ampersand (**&**) character is not needed in this circumstance to obtain a pointer value, as the variable is itself a pointer. In the example below, the variable `pStruct`, a pointer, is a parameter to function `FunctTwo`, and is passed as an argument to `FunctOne`.

The second parameter to `FunctOne` is an int. Since in function `FunctTwo`, `mValue` is a pointer to an int, the pointer must first be dereferenced using the * operator, hence the second argument in the call is `*mValue`. The third parameter to function `FunctOne` is a pointer to a long. Since `pAA` is itself a pointer to a long, no ampersand is needed when it is used as the third argument to the function.

```
int FunctOne(struct someStruct *pValue, int iValue, long *lValue)
{
    /*  do some stuff ... */
    return 0;
}

int FunctTwo(struct someStruct *pStruct, int *mValue)
{
    int j;
    long  AnArray[25];
    long *pAA;

    pAA = &AnArray[13];
    j = FunctOne( pStruct, *mValue, pAA ); /* pStruct already holds the address that the pointer will point to; there is no need
to get the address of anything.*/
    return j;
}
```

Pointers and Text Strings

Historically, text strings in C have been implemented as arrays of characters, with the last byte in the string being a zero, or the null character '\0'. Most C implementations come with a standard library of functions for manipulating strings. Many of the more commonly used functions expect the strings to be null terminated strings of characters. To use these functions requires the inclusion of the standard C header file "string.h".

A statically declared, initialized string would look similar to the following:

```
static const char *myFormat = "Total Amount Due: %d";
```

The variable `myFormat` can be viewed as an array of 21 characters. There is an implied null character ('\0') tacked on to the end of the string after the 'd' as the 21st item in the array. You can also initialize the individual characters of the array as follows:

```
static const char myFlower[] = { 'P', 'e', 't', 'u', 'n', 'i', 'a', '\0' };
```

An initialized array of strings would typically be done as follows:

```
static const char *myColors[] = {
    "Red", "Orange", "Yellow", "Green", "Blue", "Violet" };
```

The initialization of an especially long string can be split across lines of source code as follows.

```
static char *longString = "Hello. My name is Rudolph and I work as a reindeer "
    "around Christmas time up at the North Pole.  My boss is a really swell guy."
    " He likes to give everybody gifts.";
```

The library functions that are used with strings are discussed in a later chapter.

Pointers to Functions

C also allows you to create pointers to functions. Pointers to functions syntax can get rather messy. As an example of this, consider the following functions:

```
static int Z = 0;

int *pointer_to_Z(int x) {
    /* function returning integer pointer, not pointer to function */
    return &Z;
}

int get_Z(int x) {
    return Z;
}

int (*function_pointer_to_Z)(int); // pointer to function taking an int as argument and returning an int
function_pointer_to_Z = &get_Z;

printf("pointer_to_Z output: %d\n", *pointer_to_Z(3));
printf("function_pointer_to_Z output: %d", (*function_pointer_to_Z)(3));
```

Declaring a typedef to a function pointer generally clarifies the code. Here's an example that uses a function pointer, and a void * pointer to implement what's known as a callback. The `DoSomethingNice` function invokes a caller supplied function `TalkJive` with caller data. Note that `DoSomethingNice` really doesn't know anything about what `dataPointer` refers to.

```c
typedef  int (*MyFunctionType)( int, void *);      /* a typedef for a function pointer */

#define THE_BIGGEST 100

int DoSomethingNice( int aVariable, MyFunctionType aFunction, void *dataPointer )
{
    int rv = 0;
    if (aVariable < THE_BIGGEST) {
       /* invoke function through function pointer (old style) */
       rv = (*aFunction)(aVariable, dataPointer );
     } else {
         /* invoke function through function pointer (new style) */
       rv = aFunction(aVariable, dataPointer );
    };
    return rv;
}

typedef struct {
    int    colorSpec;
    char   *phrase;
} DataINeed;

int TalkJive( int myNumber, void *someStuff )
{
    /* recast void * to pointer type specifically needed for this function */
    DataINeed *myData = someStuff;
    /* talk jive. */
    return 5;
}

static DataINeed sillyStuff = { BLUE, "Whatcha talkin 'bout Willis?" };

DoSomethingNice( 41, &TalkJive,  &sillyStuff );
```

Some versions of C may not require an ampersand preceding the `TalkJive` argument in the `DoSomethingNice` call. Some implementations may require specifically casting the argument to the `MyFunctionType` type, even though the function signature exacly matches that of the typedef.

Function pointers can be useful for implementing a form of polymorphism in C. First one declares a structure having as elements function pointers for the various operations to that can be specified polymorphically. A second base object structure containing a pointer to the previous structure is also declared. A class is defined by extending the second structure with the data specific for the class, and static variable of the type of the first structure, containing the addresses of the functions that are associated with the class. This type of polymorphism is used in the standard library when file I/O functions are called.

A similar mechanism can also be used for implementing a state machine in C. A structure is defined which contains function pointers for handling events that may occur within state, and for functions to be invoked upon entry to and exit from the state. An instance of this structure corresponds to a state. Each state is initialized with pointers to functions appropriate for the state. The current state of the state machine is in effect a pointer to one of these states. Changing the value of the current state pointer effectively changes the current state. When some event occurs, the appropriate function is called through a function pointer in the current state.

Practical use of function pointers in C

Function pointers are mainly used to reduce the complexity of switch statement. Example with switch statement:

```c
#include <stdio.h>
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);
int main()
{
    int i, result;
    int a=10;
    int b=5;
    printf("Enter the value between 0 and 3 : ");
    scanf("%d",&i);
    switch(i)
    {
        case 0:  result = add(a,b); break;
        case 1:  result = sub(a,b); break;
        case 2:  result = mul(a,b); break;
        case 3:  result = div(a,b); break;
    }
}
int add(int i, int j)
{
    return (i+j);
}
int sub(int i, int j)
{
    return (i-j);
}
 int mul(int i, int j)
{
    return (i*j);
}
int div(int i, int j)
{
    return (i/j);
}
```

Without using a switch statement:

```
#include <stdio.h>
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);
int (*oper[4])(int a, int b) = {add, sub, mul, div};
int main()
{
    int i,result;
    int a=10;
    int b=5;
    printf("Enter the value between 0 and 3 : ");
    scanf("%d",&i);
    result = oper[i](a,b);
}
int add(int i, int j)
{
    return (i+j);
}
int sub(int i, int j)
{
    return (i-j);
}
int mul(int i, int j)
{
    return (i*j);
}
int div(int i, int j)
{
    return (i/j);
}
```

Function pointers may be used to create a struct member function:

```
typedef struct
{
    int (*open)(void);
    void (*close)(void);
    int (*reg)(void);
} device;

int my_device_open(void)
{
    /* ... */
}

void my_device_close(void)
{
    /* ... */
}
```

```
void register_device(void)
{
    /* ... */
}

device create(void)
{
    device my_device;
    my_device.open = my_device_open;
    my_device.close = my_device_close;
    my_device.reg = register_device;
    my_device.reg();
    return my_device;
}
```

Use to implement this pointer (following code must be placed in library).

```
static struct device_data
{
    /* ... here goes data of structure ... */
};

static struct device_data obj;

typedef struct
{
    int (*open)(void);
    void (*close)(void);
    int (*reg)(void);
} device;

static struct device_data create_device_data(void)
{
    struct device_data my_device_data;
    /* ... here goes constructor ... */
    return my_device_data;
}

/* here I omit the my_device_open, my_device_close and register_device functions */

device create_device(void)
{
    device my_device;
    my_device.open = my_device_open;
    my_device.close = my_device_close;
    my_device.reg = register_device;
    my_device.reg();
    return my_device;
}
```

Examples of pointer constructs

Below are some example constructs which may aid in creating your pointer.

```
int i;          // integer variable 'i'
int *p;         // pointer 'p' to an integer
int a[];        // array 'a' of integers
int f();        // function 'f' with return value of type integer
int **pp;       // pointer 'pp' to a pointer to an integer
int (*pa)[];    // pointer 'pa' to an array of integer
int (*pf)();    // pointer 'pf' to a function with return value integer
int *ap[];      // array 'ap' of pointers to an integer
int *fp();      // function 'fp' which returns a pointer to an integer
int ***ppp;     // pointer 'ppp' to a pointer to a pointer to an integer
int (**ppa)[];  // pointer 'ppa' to a pointer to an array of integers
int (**ppf)();  // pointer 'ppf' to a pointer to a function with return value of type integer
int *(*pap)[];  // pointer 'pap' to an array of pointers to an integer
int *(*pfp)();  // pointer 'pfp' to function with return value of type pointer to an integer
int **app[];    // array of pointers 'app' that point to pointers to integer values
int (*apa[])[]; // array of pointers 'apa' to arrays of integers
int (*apf[])(); // array of pointers 'apf' to functions with return values of type integer
int ***fpp();   // function 'fpp' which returns a pointer to a pointer to a pointer to an int
int (*fpa())[]; // function 'fpa' with return value of a pointer to array of integers
int (*fpf())(); // function 'fpf' with return value of a pointer to function which returns an integer
```

sizeof

The sizeof operator is often used to refer to the size of a static array declared earlier in the same function.

To find the end of an array (example from wikipedia:Buffer overflow):

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

int main(int argc, char *argv[])
{
  char buffer[10];
  if (argc < 2)
  {
    fprintf(stderr, "USAGE: %s string\n", argv[0]);
    return 1;
  }
  strncpy(buffer, argv[1], sizeof(buffer));
  buffer[sizeof(buffer) - 1] = '\0';
  return 0;
}
```

To iterate over every element of an array, use

```
 #define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

 for( i = 0; i < NUM_ELEM(array); i++ )
 {
     /* do something with array[i] */
     ;
 }
```

Note that the `sizeof` operator only works on things defined earlier in the same function. The compiler replaces it with some fixed constant number. In this case, the `buffer` was declared as an array of 10 char's earlier in the same function, and the compiler replaces `sizeof(buffer)` with the number 10 at compile time (equivalent to us hard-coding 10 into the code in place of `sizeof(buffer)`). The information about the length of `buffer` is not actually stored anywhere in memory (unless we keep track of it separately) and cannot be programmatically obtained at run time from the array/pointer itself.

Often a function needs to know the size of an array it was given -- an array defined in some other function. For example,

```
/* broken.c - demonstrates a flaw */

#include <stdio.h>
#include <string.h>
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

int sum( int input_array[] ){
  int sum_so_far = 0;
  int i;
  for( i = 0; i < NUM_ELEM(input_array); i++ ) // WON'T WORK -- input_array wasn't defined in this function.
  {
    sum_so_far += input_array[i];
  };
  return( sum_so_far );
}

int main(int argc, char *argv[])
{
  int left_array[] = { 1, 2, 3 };
  int right_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
  int the_sum = sum( left_array );
  printf( "the sum of left_array is: %d", the_sum );
  the_sum = sum( right_array );
  printf( "the sum of right_array is: %d", the_sum );

  return 0;
}
```

Unfortunately, (in C and C++) the length of the array cannot be obtained from an array passed in at run time, because (as mentioned above) the size of an array is not stored anywhere. The compiler always replaces sizeof with a constant. This sum() routine needs to handle more than just one constant length of an array.

There are some common ways to work around this fact:

- Write the function to require, for each array parameter, a "length" parameter (which has type "size_t"). (Typically we use sizeof at the point where this function is called).
- Use of a convention, such as a null-terminated string to mark the end of the array.
- Instead of passing raw arrays, pass a structure that includes the length of the array (such as ".length") as well as the array (or a pointer to the first element); similar to the `string` or `vector` classes in C++.

```
/* fixed.c - demonstrates one work-around */

#include <stdio.h>
#include <string.h>
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

int sum( int input_array[], size_t length ){
  int sum_so_far = 0;
  int i;
  for( i = 0; i < length; i++ )
  {
    sum_so_far += input_array[i];
  };
  return( sum_so_far );
}

int main(int argc, char *argv[])
{
  int left_array[] = { 1, 2, 3, 4 };
  int right_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
  int the_sum = sum( left_array, NUM_ELEM(left_array) ); // works here, because left_array is defined in this function
  printf( "the sum of left_array is: %d", the_sum );
  the_sum = sum( right_array, NUM_ELEM(right_array) ); // works here, because right_array is defined in this function
  printf( "the sum of right_array is: %d", the_sum );

  return 0;
}
```

It's worth mentioning that sizeof operator has two variations: `sizeof (`*`type`*`)` (for instance: `sizeof (int)` or `sizeof (struct some_structure)`) and `sizeof` *`expression`* (for instance: `sizeof some_variable.some_field` or `sizeof 1`).

External Links

- "Common Pointer Pitfalls" (http://www.cs.cf.ac.uk/Dave/C/node10.html#SECTION001080000000000000000) by Dave Marshall