C for Java Programmers

J. Maassen

Contents

Chapter 1

Introduction

This reader is designed to help Java programmers learn the C programming language. The focus is on the differences between C and Java. We assume the reader has reasonable programming skills.

The practical examples presented in this reader are based on a Unix environment and use the GNU C Compiler (*gcc*) and related tools. They can be downloaded from `http://gcc.gnu.org/onlinedocs/gcc.html`.

The text of Chapter 5 is taken from *Ted Jensen's Tutorial on Pointers and Arrays in C*, which can be downloaded from `http://home.netcom.com/~tjensen/ptr/cpoint.htm`.

1.1 History of C

The C programming language was developed in the early 1970's by Brian Kernighan and Dennis M. Ritchie. It was based on the B [1] programming language, which in turn was based on BCPL [3]. The following text is taken from the paper "The Development of the C Language" by Dennis M. Ritchie [4].

C came into being in the years 1969-1973, in parallel with the early development of the Unix operating system; the most creative period occurred during 1972. Another spate of changes peaked between 1977 and 1979, when portability of the Unix system was being demonstrated. In the middle of this second period, the first widely available description of the language appeared: The C Programming Language, often called the 'white book' or 'K&R' [Kernighan 78]. Finally, in the middle 1980s, the language was officially standardized by the ANSI X3J11 committee, which made further changes. Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, the language was almost exclusively associated with Unix; more recently, its use has spread much more widely, and today it is among the languages most commonly used throughout the computer industry.

1.2 Recommended Literature

Many book about the C programming languages exists. A few of the more popular ones are:

```
The C Programming Language, Second Edition
by Brian W. Kernighan and Dennis M. Ritchie.
Prentice Hall, Inc., 1988.
ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hard back).

The C Puzzle Book, Revised edition
by Alan R. Feuer
Addison-Wesley Pub Co., October 1998.
ISBN 0201604612

A Book on C, 4th edition
by Al Kelley, Ira Pohl
Addison-Wesley Pub Co., January 1998.
ISBN 0201183994
```

There are also a number of books on C specifically written for Java programmers, for example:

```
C for Java Programmers
Tomasz Muldner, Acadia University, Canada
ISBN 0-201-70279-7
```

Before buying a book, try to find some reviews first (e.g., *Customer Reviews* at amazon.com) to determine if it fits your needs and is worth the money. You can also find a lot of C programming courses, references guides, and FAQs on the Internet. Here are a few links (in no particular order).

```
http://www.strath.ac.uk/CC/Courses/NewCcourse/ccourse.html
http://www.acm.uiuc.edu/webmonkeys/book/c_guide/index.html
http://www.cs.ntu.edu.au/sit/resources/cprogram/default.htm
http://www.graylab.ac.uk/doc/tutorials/C/
http://www.1001tutorials.com/c/index.shtml
http://www.eskimo.com/~scs/C-faq.top.html
```

1.3 Differences between C and Java (overview)

Although the syntax of Java and C are very similar, they are very different languages. The following table shows some of the major differences:

Java	C
Object-Oriented	Procedural
Interpreted	Compiled
Memory Management	No Memory Management
References	Pointers
Exceptions	Error Codes

Object-Oriented vs. Procedural One of the largest differences between Java and C is the use of a different programming *paradigm*. Java is an Object-Oriented language. A Java program consists of a collection of objects. These objects contain the data used in the program, and have methods to perform operations on this data.

The C language is *procedural*. A C program consists of a collection of *procedures* (or *functions*). The data used by the program can be put into local variables (inside of a function) or global variables (outside of functions) There is no notion of objects in C. Just like in Java, there is a special *main* function, which is used to start the program.

Interpreted vs. Compiled Java is an interpreted language. Java source code is transformed to *bytecode*, which is then loaded by a program called an interpreter. This program then 'executes' each of the bytecode instructions one by one, translating them into something the machine understands.

C programs are *compiled*. Instead being translated to some intermediate format (like bytecode) it is translated directly into machinecode. This machinecode is directly executed by the processor.

Memory Management vs. No Memory Management In Java, the memory management is done automatically by the system. New objects can be created using the *new* keyword. When objects are no longer used (i.e., no longer have any references pointing to them) they are removed by the garbage collector.

In C, the programmer has to do his own memory management. Using the keyword *sizeof* and the library calls *malloc* and *free*, blocks of memory can be allocated and freed. This will be explained further in Section 5.10.

References vs. Pointers A reference in Java is a special variable which references (points-to) an object. Only objects can be referenced. For example, it is not possible to have a reference to an *int*.

Pointers in C are in some ways similar to references in Java (they point to things), but in many ways they are very different. Chapter 5 will explain pointers in more detail.

Exceptions vs. Error Codes Whenever an error occurs in Java, an *exception* is thrown. C has no exceptions. A function either returns some error code (when an error is expected), or your program crashes (usually long after the error has occurred). Section 4.6 describes how you can find and prevent errors.

Chapter 2

Overview of C

2.1 Example program

Lets start out with a simple example program: `myprogram.c`.

```
#include <stdio.h>

double value;

/* This is a comment */

int main(void)
{
    int local = 0;

    value = 0.42;

    printf("local = %d value = %f\n", local, value);

    return 0;
}
```

The program starts with the line '`#include <stdio.h>`', which is actually not C-code, but a *preprocessor* directive. The preprocessor is a special program which pre-processes the C program before it is compiled. All statements beginning with '`#`' are preprocessor directives. The pre-processor will be explained in Section 4.1.

The purpose of the '`#include <stdio.h>`' statement is similar to an '`import`' in Java. It imports a *header file* called '`stdio.h`' into this program. Header files contain descriptions (or *prototypes*) of functions and types which are implemented and defined somewhere else. They usually have the extension '`.h`'.

By importing the header file '`stdio.h`', the functions and type described in that file can be used in this file (in this case we are interested in using the '`printf`' function).

The next line '`double value;`' defines a global variable. Global variables are variables which are defined outside the scope of a function. They exist throughout the lifetime of the program (they are created when the program starts and destroyed when the program exits). Because they are global, the can be 'seen' and used in every function in this file.

After the comment '`/* This is a comment */`' (which is similar to a Java comment) the special startup function '`int main(void)`' is declared, which returns an '`int`' result, and takes no parameters. In C programs, it is customary to return an '`int`' result from the '`main`' function. A result of 0 indicates that no error has occurred. Any other value indicates an error. When a function is C takes no parameters, this is indicated by the `(void)` parameter list.

The '`int local = 0;`' line declares a local variable in the main function. The rules in C for declaring local variables are a little different from Java. All local variables must be declared at the beginning of the function. Declaring them at a later point will result in a compile time error.

After declaring the local variable, a value is assigned to the global variable in the statement '`value = 0.42;`'. The values of the variables are then printed to the screen using the '`printf("local = %d value = %f\n", local, value);`' statement. The '`printf`' function will be explained in more detail in Section 4.3.1 Finally, a value of 0 is returned (`return 0;`) to indicate that the program has finished without any errors.

We can now compile the C program using the following command:

```
gcc myprogram.c
```

In this example, the C compiler is called '`gcc`' (this stands for *GNU C Compiler*. The result is an executable file, called '`a.out`' which we can run.

```
./a.out
```

Our program then produces the following output:

```
local = 0 value = 0.420000
```

2.2 Keywords

There are 32 reserved keywords in the C language:

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Most of these (like '`switch`' and '`return`') are well known from Java and will not be explained in detail. The ones that are not used in Java will be explained in the following sections (except for the ones like '`goto`' and '`auto`' that you will never need).

2.2.1 Build in types

The C language has the following build in types : '`char`, '`short`', '`int`', '`long`', '`float`', and '`long double`'. There is also an unofficial '`long long`' type, which is supported by most compilers. Contrary to the primitive types in Java, most of the types in C have no fixed size. For example: an '`int`' in Java is always 32 bits, in C, however, the size of an '`int`' is specified as *the natural size of a word* on a processor. As a result, an '`int`' may be 16 bits on older machines (e.g., using the 16 bit Motorola 68000 processor), 32 bits on current machines (e.g., using an Intel Pentium) and 64 bits on high-tech machines (e.g., a 64 bit DEC Alpha processor). An exotic machine, like the Honeywell 6000, even had a 36 bits '`int`'.

The following table shows the sizes of Java's primitive types and 'reasonable' values for C's types.

type	Java	C
char	16 bits	8 bits
short	16 bits	16 bits
int	32 bits	16, 32 or 64 bits
long	64 bits	32 or 64 bits
float	32 bits	32 bits
double	64 bits	64 bits
boolean	1 bit	(use int)
byte	8 bits	(use char)
long long		64 bits (unofficial)
long double		80, 96 or 128 bits

Some of Java's primitive types do not exist in C. For example, there is no '`boolean`' type in C, normally an '`int`' is used instead (but you can use another type if you like). An value of 0 is interpreted as *false*, any other value is interpreted as *true*. As a result, the '`if`', '`while`' and '`for`' statements expect an '`int`' instead of a '`boolean`'. For example, the C statement '`if (1)`' is equal to the Java statement '`if (true)`'. The following code shows some examples. It also shows some mistakes which are often made in C.

```
void example(void) {
    int y;
    int x = 100;

    if (x == 4) { }     /* not taken */

    if (x) { } /* taken */

    while (x) { x--; }  /* repeats 100 times until x is 0 */

    /* MISTAKES */
    if (x = 1) { }      /* We forgot an '=' here ! Now 1 is
                         * assigned to x, the result is 1,
                         * so this if is taken !
                         * This is a SYNTAX error in Java,
                         * but allowed in C
                         */

    while (y) { y--; }  /* We forgot to initialize y !
                         * This is a COMPILE error in Java,
                         * but is allowed in C
                         */
}
```

In Java, all primitive types except '`boolean`' are *signed* (they can have both positive and negative values). The C language also support *unsigned* types (they can only have positive values). The following example shows three ways of defining an integer.

```
         int i1; /* range -2,147,483,648 to 2,147,483,647 */
  signed int i2; /* range -2,147,483,648 to 2,147,483,647 */
unsigned int i3; /* range 0 to 4,294,967,295 */
```

Note that a '`signed int`' is the same as an '`int`'. Floating point values are always signed.

2.2.2 Identifiers

In C, the following rules apply to identifiers (the names you give to functions and variables). An identifier,

- consists of any uppercase or lowercase characters, numerical digits (0 through 9), and the underscore character (_).

- may not begin with a numerical digit or underscore.
- is case sensitive.

2.2.3 Variables

We will now briefly describe the properties of variables in C. (we have already seen some examples).

Global Variables When a variable is declared outside of a function it is called a *global* variable. Global variables can be 'seen' and used by all the functions of the file (this is called *file scope*) and exist throughout the lifetime of the program. They are created when the program starts and destroyed when the program exits. There are a number of keywords which modify the behavior of global variables:

- To use a global variable declared in another file, the variable can be re-declared using the `extern` keyword. For example,

  ```
      extern int value;
  ```

  The '`int value`' variable will not be created (in this file). The declaration only tells the compiler that there is a '`int value`' variable *in some other file*.

- Constants can be defined by placing the keyword `const` in front of the variable declaration.

  ```
      const int size = 42;
  ```

  This is similar to the `final` keyword in Java. Constants in C are usually defined in another way, using the `#define` preprocessor directive (see Section 4.1). The `const` keyword can also be used for local variables.

- By placing the `static` keyword before a global variable, the scope of the variable (i.e., the places it can be used) is reduced to the file it is declared in.

  ```
      static int large;
  ```

  The variable `large` can now only be used inside the file it is declared in, even if it is declared `extern` in some other file. By using the `static` keyword, global variables in different files can have the same name without interfering with each other.

  This is very different from the `static` keyword in Java !

Local Variables When a variable is declared in a function it is called a *local variable*. Just like in Java, local variables can only be used in the function they are declared in. Each time the function is called, the local variables of that function are created. When the function returns, the local variables are destroyed. Unlike Java, the local variables of a C function, must be declared at the beginning of the function, before the other statements. There are a number of keywords which modify the behavior of local variables.

- By placing the `static` keyword before a local variable, the lifetime of the variable is increased to the lifetime of the program. For example:

  ```
      void function(void) {
          static int count = 0;
          count++;
      }
  ```

  The variable `count` will not be destroyed when the function returns. When the function is called for the first time, the `count` variable is initialized to `0` (`static int count = 0`). The function then increments the count variable to `1` (`count++`) and returns. The next time the function is called, the `count` variable still exists (and has the value `1`). It will not be initialized again. The function the now increments the count variable again. Thus, `count` count the number of times the function is called. Static local variables can be seen as global variables which can only be used in a single function.

- Placing the `register` keyword before a local variable, gives the compiler a hint that this variable is important for performance and must be accessed as quickly as possible. The compiler will then try to put this variable into a hardware register, where it can be accessed quickly (it may not succeed though!). An example,

  ```
      void function(void) {
          register int important = 0;
          volatile int magic = 0;
      }
  ```

  The opposite of `register` is the `volatile` keyword. This forces the compiler not to put the variable into a register. This useful for variables which may be changed unexpectedly (e.g., by some other thread or device). The `volatile` keyword can also be used for global variables. You will probably not need the `register` and `volatile` keywords often.

2.3 Operators

We already know most of the C operators from Java. We will therefore only briefly list them.

`!exp`	logical not	`exp++`	increment
`exp && exp`	logical and	`exp--`	decrement
`exp || exp`	logical or	`++exp`	increment
`exp == exp`	logical equals	`--exp`	decrement
`exp != exp`	logical not equals	`exp + exp`	addition
`exp < exp`	logical smaller	`exp - exp`	subtraction
`exp > exp`	logical greater	`exp * exp`	multiplication
`exp <= exp`	logical smaller or equal	`exp / exp`	division
`exp >= exp`	logical greater or equal	`exp % exp`	modulo
`exp = exp`	assignment	`~exp`	bitwise not
`exp += exp`	addition assignment	`exp & exp`	bitwise and
`exp -= exp`	subtraction assignment	`exp | exp`	bitwise or
`exp *= exp`	multiplication assignment	`exp ^ exp`	bitwise xor
`exp /= exp`	division assignment		
`exp %= exp`	modulo assignment	`exp >> exp`	shift right
`exp >>= exp`	shift right assignment	`exp << exp`	shift left
`exp <<= exp`	shift left assignment		
`exp &= exp`	bitwise and assignment		
`exp |= exp`	bitwise or assignment		
`exp ^= exp`	bitwise xor assignment		
`exp ? exp : exp`	conditional operator		

Note that the logical operators in Java use boolean values. In C, integer values are used.

2.4 Statements

The `if`, `while`, `for` and `switch` statements in C behave in almost the same way as their Java counterparts. The differences are:

- The expressions in the `if`, `while` and `for` statements do not use boolean values. Any type will do. A value of `0` is evaluated as `false`, any non-zero value is evaluated as `true`.
- The counter variable used in the `for` statement must be declared in the beginning of the function, like other local variables.

Chapter 3

More about types and functions

3.1 Arrays

In Java arrays are a special type of objects. An array variable in Java is a reference to an array object. The memory required for the array object is allocated at run time, using the `new` operator. As a result, when the array variable is declared it is not necessary to specify its size. This can be done when the array is created. The following statements are Java code.

```
/* This is Java code */
int [] a1, a2;
a1 = new int[8];

int [] a3 = { 1, 2, 3 };
a2 = a3;
a2[10] = 5;  /* throws an exception */
```

Although arrays in C look similar to the arrays used in Java, the rules are different. In C, you can declare an array like this:

```
type identifier[size];             /* normal array           */
type identifier[size1] ... [sizeN]; /* N dimensional array    */
type identifier[] = { value-list }; /* array with initializer */
```

The `type` is the element type of the array, `identifier` is its name, and `size` the number of elements it contains. Note that when an array is declared, the array brackets must come *after* the name of the array (this is different in Java).

An important difference between Java an C is that arrays in C are not references, but just a block of memory with a name attached to it. As with other variables, the amount of memory needed for an array is determined at compile time, not at runtime ! As a result, when an array is declared in C, it size must be must be immediately specified (or the compiler must be able to calculate it).

After the array is created, its size cannot be changed, because the memory it uses was reserved at compile time. For the same reason, the array can not be reassigned to point to a new array. (it is not a reference, just a named block of memory). When you create an array as a local variable, it will only exist as long as the function exists. So it is not possible to return an array as a function result.

It would seem that arrays in C are not as flexible as the arrays we now from Java. Fortunately, there is a (more flexible) way to create arrays, by using the `malloc` call. This will be described in Chapter 5.

Another difference between Java and C, is that there is no array bounds checking in C. Since the array is just a block of memory with a name, it has no `length` field or special methods to check if an array access is within the bounds. As a result, writing at index 10, 1000 or -5 of an array of length 9 is 'allowed' in C (although your program will probably crash). The following code shows some examples of correct an incorrect use of arrays in C.

```
int a1[5];            /* OK                                  */
int a2[] = { 1, 2, 3 };/* OK (size calculated by compiler)     */

int a5[9];
a5[10]   = 7;         /* OOPS, C does not do any array bound  */
a5[1000] = 9;         /* checking. It accepts these statements */
a5[-5]   = 5;         /* even though they are clearly WRONG   */

int a3[];             /* ERROR, needs size                    */
a3 = a2;              /* ERROR, cannot assign arrays          */
int [] a4 = { 1, 2 }  /* ERROR, wrong notation                */

int [] function(void) {/* ERROR, can not return an array       */
    int array[5];
    return array;
}

void foo(int a[]) {   /* OK, can have an array as a parameter  */
    a[3] = 0;
}

void bar(void) {
    int a[5];         /* OK, can pass an array as a parameter  */
    foo(a);           /* because bar() exists longer than foo()*/
}
```

3.2 Strings

A String in Java is a special type of object. A String object contains an array of characters, which contains the actual text of the String. The String objects also contains methods which allow the programmer to perform operations on the String.

In C, a string is simply an array of characters. Strings in C are expected to end with a special character '`\0`' (called *nul*). This character is used to find the end of the string. As a result, C-strings are always at least one character longer than the text they contain. Strings can be initialized in a number of ways:

```
char name0[6];
name0[0] = 'J';
name0[1] = 'a';
name0[2] = 's';
name0[3] = 'o';
name0[4] = 'n';
name0[5] = '\0';
char name1[] = { 'J', 'a', 's', 'o', 'n', '\0' };
```

These are normal ways of initializing an array. A string specific form of initialization can also be used.

```
char name2[6] = "Jason";
char name3[] = "Jason";
char name4[100] = "Not 100 characters long!";
```

The string `"Jason"` will be translated by the compiler into special code which initializes the array. Note that it is not necessary to explicitly write down the '`\0`' character, this is done for you. As example `name3` shows, it is also not necessary to specify the size of the string if it is immediately initialized. The compiler will automatically generate a char array which is large enough. It is also possible to specify that the char array is larger than the string, as example `name4` shows.

Functions to perform operations on strings can be imported using the header file `string.h`. Strings will be explained in more detail in Chapter 5.

3.3 Enumerations

Using an *enumeration*, a series of integer constants can be created. An enumeration is created as follows:

```
enum identifier {list};
```

The `identifier` is a name used for the enumeration and is optional. The `list` is a list of constant integer variables to be created. The first variable is given the value of 0. Each variable is given the value of the previous variable plus 1. It is also possible to specify your own values. If you have given a name to your enumeration, it is possible to create a variable of the enumeration type.

```
enum enumeration-name variable-name1, variable-name2, ...;
```

Some examples:

```
/* Creates 3 constants. aap is 0, noot is 1, and mies is 3. */
enum {aap, noot, mies};

/* Creates 5 constants with identifier 'workdays'.
 * monday = 42, tuesday = 55, wednesday = 56, etc.
 */
enum workdays {monday = 42, tuesday = 55, wednesday,
               thursday, friday };

/* Create a variable of the type 'workdays' */
enum workdays today;
today = tuesday;
today = friday;
```

3.4 Structures

Structures provide a way to group a number of variables together. It is similar to a very simple Java object (without any methods). A structure can be defined as follows:

```
struct identifier {
    type variable_names;
    type variable_names;
    ...
} structure-variables,...;
```

The `identifier` is a name used for the structure and is optional. This name can later be used to create variables of this structure type. The structure contains a number of variables, each with a `type` and an `variable_name`. By specifying a number of `structure-variables` it is possible to immediately create variable of the structure type. If the structure was given name, new variables of the structure type can be created like this:

```
struct structure-name variable-name1, variable-name2, ... ;
```

To access a variable in the structure, you select one of the fields using a *record selector* '.'. Here are some examples:

```
/* Create a nameless struct and create two variables */
struct {
    int val1, val2;
    double val3;
} s1, s2;

s1.val1 = 5;
s2.val3 = 4.6;

/* Create a named struct and create the variables afterwards */
struct ComplexNumber {
    double real, imag;
};

struct ComplexNumber num;
num.real = 2.5;
num.imag = 0.3;

/* You can even use structs in structs ! */
struct Parameters {
    ComplexNumber complex;
    double value;
};

struct Parameters param;
param.complex.real = 3.6;

/* ... and create arrays of structs ! */
struct Parameters wow[5];

wow[3].complex.real = 3.6;
```

3.5 Unions

Unions look very similar to structures. The difference between them is, that all variables in a union use the same memory location. When a union variable is created, enough space is allocated for the largest variable in the union. All other variables share the same memory. Unions are defined like this:

```
union identifier {
    type variable_names;
    type variable_names;
    ...
} union-variables,...;

union union-name variable-name1, variable-name2, ... ;
```

Unions are used when a variable can have values of different types. Here are some examples:

```
/* Create a union of double and int. The size will be 64 bits */
union MyUnion {
    int i_value;
    double d_value;
};

union MyUnion u;
u.i_value = 6;
u.d_value = 5.4;  /* Overwrites the i_value !! */

/* You can also use a struct in a union, the size will be the
 * size of the largest struct, in this case 2 doubles = 128 bits
 */
union Value {
    ComplexNumber complex;
    double normal;
};

union Value val;

val.normal = 5.7;

val.complex.real = 7.9; /* These two exist at the same time ! */
val.complex.imag = 9.8; /* But overwrite the 'normal' field ! */
```

3.6 Defining types

Using the `typedef` keyword, you can define new names for types. This allows you to come up with better names for your data structures (and you don't have to type `struct` or `union` every time create a variable). Some examples of `typedef`:

```
/* Define a new type 'byte', which is actually a char */
typedef char byte;
byte b = 123;

/* Define a new type complex
 * (the ComplexNumber struct of the previous section)
 */
typedef struct ComplexNumber complex;

complex var;
var.real = 5.9;
var.imag = 0.1;

/* Define a new type int_or_double
 * (the MyUnion union of the previous section)
 */
typedef union MyUnion int_or_double;

int_or_double x;
x.i_value = 5;
x.d_value = 6.4; /* Overwrites the i_value !! */
```

3.7 Casting

Like Java, the C language supports *casting* one type to another. For example:

```
int i;
double d = 6.7;

i = (int) d;
```

This will cast the `double` value 6.7 to the `int` value 5. Casting will come in handy when working with arrays and function pointers (see Chapter 5).

3.8 Functions

Functions in C are a little different from Java methods. A function is declared in the following way:

```
type identifier(parameter-list) { function-body }
```

The `type` is the type that the function returns (this can not be an array type, see

Section 3.1). The `identifier` is the name of the function. The `parameter-list` is the list of parameters that the function takes separated by commas. If the function does not have any parameters, then the `parameter-list` should be empty (`int main()`), or use `void` (`int main(void)`). When you pass an array as a parameter to a function, is is automatically converted to a pointer. This will be explained in Chapter 5.

You can also create a C function with a variable number of parameters. The parameter list is must then be terminated like this:

```
int foo(int value, double d, ...) { }
```

The '`...`' indicates that any number of parameters may follow. To access the extra parameters, you need functions which can be imported from `stdarg.h`. The `printf` function is an example of a function with a variable number of arguments (see Section 4.3.1).

If a function is used before it is defined, a *prototype* must be made, so the compiler knows what the function looks like. Prototyping normally occurs at the beginning of the source code or inside of header files (see Section 4.2), and is done in the following manner:

```
type identifier(parameter-type-list);
```

The `type` and `identifier` must be exactly the same as the actual function definition. The names of the parameters do not have to be given here (although they may be given for the sake of clarity). For example:

```
int max(int, int); /* prototype */

void example(void)
{
    int result = max(5, 8);
}

int max(int one, int two)
{
    return (one < two ? two : one);
}

int main(void) {
    example();
}
```

Functions in C are first class entities. A variables of the type 'function' can be created, pointers to functions can be passed as parameters, and returned as a result values. This will be explained in Section 5.11

3.8.1 The main function

Like Java, C programs us a special *main* function to start the program. The main function looks like this:

```
int main(void) { ... }
int main(int argc, char** argv) { ... }
```

We have already seen the first form in example programs. The second form can be used if your program needs command line arguments. It has two parameters. The first, `argc`, contains the number of command line arguments passed to your program. The second, `argv`, is an array of strings. One string for each of the arguments. The following program prints it command line arguments:

```
int main(int argc, char **argv)
{
    int i;
    for (i=0;i<argc;i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}

/* compile and run */
gcc example.c
./a.out aap noot mies

/* Output*/
a.out
aap
noot
mies
```

Note that the name of the program, `a.out`, is passed as the first command line argument.

Chapter 4

Creating C programs

We have now shown you some simple examples of programming in the C language. However, to be able to create a real C program, you must know more about the preprocessor, libraries, the compiler, and how to create header files.

4.1 Preprocessor

The C preprocessor (*cpp*), is a program which filters your source code before it is compiled. It is invoked automatically by the compiler so you will not need to call it yourself. The preprocessor reads all of your C code and reacts to *preprocessor directives*. These directives can be recognized because they start with a '`#`' character. It changes your program according to the directives you use and produces a new copy, which can then be compiled. You can use the following preprocessor directives:

4.1.1 Defining things

The `#define` directive can be used to define constants and *macros*.

```
#define identifier replacement-code
#define identifier(parameter-list) (replacement-text)
#undef identifier
```

After a `#define`, the preprocessor will replace all occurrences of `identifier` with the `replacement-code`. If you give a `(parameter-list)` to `#define`, all the parameters will be inserted into the `replacement-text` (this is called a *macro*). The `#undef` can be used to remove a previous `#define`.

```
#ifdef identifier
<code>
#else (optional)
<code>
#endif

#ifndef identifier
<code>
#else (optional)
<code>
#endif
```

The `#ifdef`, `#ifndef`, and `#else` can be used to check if some `identifier` is defined. This allows you to conditionally compile certain lines of code. The following code shows an example:

```
#define LOOPS 100                 /* constant */
#define MAX(A, B) (A < B ? B : A) /* macro */
#define DEBUG2

void function(void) {
    int i, j;
#ifdef DEBUG1
    /* will only be executed if DEBUG1 is defined */
    printf("starting loop\n");
#endif
    for (i=0;i<LOOPS;i++) {
#ifdef DEBUG2
        /* will only be executed if DEBUG2 is defined */
        printf("in loop %d\n", i);
#endif
        j = MAX(i, j);
    }
}

/* This will be converted to */
void function(void) {
    int i, j;

    for (i=0;i< 100 ;i++) {
        printf("in loop %d\n", i);
        j = (i < j ? j : i) ;
    }
}
```

4.1.2 Testing things

The #if, #elif, #else, #endif preprocessing directives can be used to conditionally compile parts of the source code. They have the following syntax:

```
#if const_exp
#else
#endif

#if const_exp
#elif const_exp
#endif
```

The value of the const_exp is evaluated in the same way that an if statements evaluates its expression (value 0 is *false*, other values are *true*). Note that this is a different behavior than #ifdef. The following shows an example:

```
#define FIRST  1
#define SECOND 0

void function(void)
{
#if FIRST
    printf("first\n");
#endif
#if SECOND
    printf("second\n");
#endif
}

/* This will be converted to */
void function(void)
{
     printf("first\n");
}
```

4.1.3 Including things

The #include directives can be used to import a header file into the current file.

```
#include <filename>
#include "filename"
```

The header file filename will be *copied* into the current file at the position of the #include directive. The <filename> form is used to include system libraries (like `stdio.h`), while the `"filename"` form is used to include local header files (when you have a large program split into many smaller files). For example:

```
#include <stdio.h>

void function(void) {
    printf("Hello world\n");
}

/* This will be converted to */

int printf(const char *__format, ...);
int sprintf(char *__s, const char *__format, ...);
... (lots of other function prototypes!)

void function(void) {
     printf("Hello world\n");
}
```

More information about the `cpp` preprocessor can be found at
`http://gcc.gnu.org/onlinedocs/cpp.html`.

4.2 Header files

We have already shown you some example of how you can use libraries by including header files. It is often useful to use header files when you are writing a program yourself (e.g., when you are writing a large program you may want to split it up into multiple smaller '`.c`' files). To used the types and functions of one '`.c`' file in another '`.c`' file you need to make header files. The following example shows a simple header file, `complex.h`.

```
#ifndef _COMPLEX_HEADER_FILE_
#define _COMPLEX_HEADER_FILE_

struct ComplexNumber {
    double real, imag;
};

typedef struct ComplexNumber complex;

complex c_create(double real, double imag);
complex c_add(complex c1, complex c2);
complex c_mult(complex c1, complex c2);
#endif
```

In this header file, a new `complex` type is defined, which consists of a struct of two doubles and can be used to represent a complex number.. The header file also contains prototypes of the functions `c_create`, `c_add` and `c_mult` which can be used to create, add and multiply complex numbers.

Because you are not allowed to declare the same types and functions more than once, a compile time error would occur if a header file is included multiple times. To prevent this, the header file defines the text '`_COMPLEX_HEADER_FILE_`' the first time it is included. If it is included again later, the '`#ifndef _COMPLEX_HEADER_FILE_`' check will fail, and all the text up to the '`#endif`' (the entire header file) is skipped.

```
/* This is the file complex.c */
#include "complex.h"

complex c_create(double real, double imag)
{
    /* create and return a new complex */
}

complex c_add(complex c1, complex c2)
{
    /* add two complex and return a new one */
}

complex c_mult(complex c1, complex c2);
{
    /* mul two complex and return a new one */
}
```

The `complex.c` file contains the actual implementations for the `c_create`, `c_add` and `c_mult` functions. It includes the `complex.h` file to find the definition of the `complex` type, and the prototypes of the functions. Any other file that wishes to use the `complex` type can now include `complex.h`.

4.3 Libraries

There are a large number of C-libraries available which you can use in your program. Libraries have a special file format. For example, the library 'list' would be in the file `liblist.a` or `liblist.so`. You must also tell the compiler that you want to use the library. This is explained in Section 4.5. You can use the functions in a library by including its header file. The following table list some of the most frequently used libraries:

`stdio.h`	Input/output functions
`stdlib.h`	Some standard functions and macros
`stddef.h`	Some standard definitions (types)
`math.h`	Mathematical functions
`stdarg.h`	Functions to use a variable number of parameters
`string.h`	Functions to manipulate strings
`time.h`	Functions related to time

Since we often use the `printf` function of the `stdio.h` library, we will describe this function in a little more detail. If you want more information other functions in the `stdio.h` library, or any other library, you can use the `man pages` described in the next section.

4.3.1 printf

The `printf` function can be used to print something to the screen. There are a lot of variations on `printf` which allows you to print to other destinations (for example: `fprintf` prints to a file, `sprintf` prints to a string). Only printf will be explained here. The `printf` function has the following definition:

```
int printf(const char *format, ...);
```

The '`const char *format`' means that the first argument of `printf` must be a string (called *the format string*) describing the output. For example:

```
printf("Hello world\n");              /* Output: Hello World */

int val = 5;
printf("Hello %d world\n", value);     /* Output: Hello 5 World */

char c = 'a';
printf("val = %d c = %c\n", value, c); /* Output: val = 5 c = a */

char [] world_str = "world";
printf("Hello %s\n", world+str);       /* Output: Hello World */
```

As you can see, we can directly put any text we want to print into the format string. To print a the value of a parameter, we must specify where we want to put it in the string, and what the type of the variable should be, using the `%` notation. Below are some of the types the `printf` function can handle.

`%d`	signed int
`%u`	unsigned int
`%x`	hexadecimal unsigned int
`%c`	character
`%f`	double and float
`%s`	string
`%%`	to print a `%`

The next section will describe how you can find more information about `printf` and other functions.

4.4 Getting help

If you are using a Unix system, you can get more information on C libraries and functions by using the *man* command. The man command can be used like this:

```
man -S section subject
```

The '`-S section`' part is optional (but often needed). Information on C libraries and functions can be found in section 3. The '`subject`' is the name of the library or function you want information on. For example:

```
man -S 3 printf

/* Output */

PRINTF(3)           Linux Programmer's Manual           PRINTF(3)

NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf, vfprintf,
       vsprintf, vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       etc.
```

You can also visit `http://www.acm.uiuc.edu/webmonkeys/book/c_guide/` for a description of some of the libraries in C.

4.5 Compiling

In Section 2.1 we have already shown you an example of how to compile a file:

```
gcc myprogram.c
```

The result was an executable file, called `a.out` which we could run. If we look at compiling a C program in more detail, we see that it actually consists of three steps:

1. *Preprocessing.* The '`.c`' file is preprocessed, which copies the header files into the '`.c`' file, handles `#define` and removes comments.
2. *Compiling.* The result of the preprocessor is compiled and produces a *binary* file called `myprogram.o`. This binary file contains the actual machine code of your program.
3. *Linking.* All the binary files of your program (and libraries they use) are *linked* together. The linker program combines all the binary files into one executable. To do this, it has to resolve the `extern` variables and function prototypes (For example, it *links* the call to `printf` in our example to the real `printf` function in took from the library).

Compiling a program in the way we showed in Section 2.1 only works if a program consists of a single file. If your program is split up into multiple files, each of these file must be compiled separately, and linked together afterwards.

```
gcc -c file1.c
gcc -c file2.c
gcc -c file3.c
gcc file1.o file2.o file3.o -o myprogram
```

The '`gcc -c`' command tells the compiler to compile the file, but to skip the link phase (which must wait until we have compiled all the files). The compiler will then produce a '`.o`' file for each of the '`.c`' files it compiles. When all the files are compiled, we link them together (using the last command). The compiler sees that all input files are '`.o`' files and no compiling is necessary. It than links the files together into the executable called '`myprogram`'.

The following table shows a few of the options the compiler supports:

`-c`	skip the link phase
`-I<dir>`	add the `<dir>` to the include path
`-l<lib>`	link the library `<lib>` to program
`-L<dir>`	add the `<dir>` to the library path
`-w`	give no warnings
`-Wall`	give extra warnings
`-O`	optimize the code
`-O2`	optimize the code even more
`-g`	produce debugging code
`-p or -pg`	produce profiling code

The `-I<directory-name>` option can be used to add directories to the search path for the include files. For example:

```
gcc -I/home/jason/includes -I./includes file.c
```

This command adds the directories '`/home/jason/includes`' and '`./includes`' to the search path.

Sometimes the compiler needs to be told to include a library when linking the program. For example:

```
#include <math.h>

int main(void)
{
    double d = cos(1.2);
    return 0;
}

/* compile this program */
gcc example.c

/* output */
/tmp/ccBtnHGs.o: In function ‘main’:
/tmp/ccBtnHGs.o(.text+0x16): undefined reference to ‘cos’
collect2: ld returned 1 exit status
```

When we compile the program `example.c`, the compiler gives the error 'undefined reference to ‘cos”. What is happening here is that the compiler tries to find the function `cos` during linking, but can not find it in any of the '`.o`' files it knows. This function is part of the library `libm.a`. To include this library in the linking you must use the `-lm` command:

```
gcc example.c -lm
```

The program is now compiled and linked successfully. Note that the '`lib`' and

'.a' part of the library name are not specified (only the 'm' part matters). Most of the functions you will need are in the library `libc.a`. This library is always linked with your program automatically.

More information about the command line options of `gcc` can be found at `http://gcc.gnu.org/onlinedocs/gcc_3.html`

4.5.1 Makefiles

When you are writing large multi-file programs, compiling can become quite complicated. Just imagine having to compile hundreds of '.c' files by hand, and trying to keep track of the dependencies between those files. The *make* utility can help you manage these large programs. To use *make* the programmer has to write a so called `Makefile`. This `Makefile` describes which files are part of the program, the dependencies between them, what complier to use, etc. The following text shows an example `Makefile`.

```
# This is a comment !
#
# 'myprogram' is built from file1.c file2.c and file3.c

CC = gcc
CFLAGS = -Wall
OBJS = file1.o file2.o file3.o

myprogram: $(OBJS)
        $(CC) -o myprogram $(OBJS)

# ^^^ This space must be a TAB!!.
```

The `Makefile` starts by defining a number of variables:

- `CC` defines the name of the compiler (`gcc`).
- `CFLAGS` defines what flags the compiler should use (`-Wall`).
- `OBJS` defines files are part of the program (`file1.o file2.o file3.o`).

The next line of the `Makefile`, '`myprogram: $(OBJS)`', gives a dependency rule. It says: 'to make myprogram, you must first make OBJS'. The *make* utility will then look at `OBJS`, see that it consists of three '`.o`' files, and find the '`.c`' files it needs to compile to generate these '`.o`' files.

After the '`.o`' files are generated (using the compiler described in `CC`), *make* analyses the next line in the `Makefile`. This line, '`$(CC) -o myprogram $(OBJS)`' tells *make* that is should call the compiler again, passing it the parameters `-o myprogram $(OBJS)`. This command will link all the '`.o`' files into a program called `myprogram` (note that this line starts with a tab). We can now use this `Makefile` to compile our program:

```
make myprogram

/* Output */
gcc -Wall -c file1.c
gcc -Wall -c file2.c
gcc -Wall -c file3.c
gcc -o myprogram file1.o file2.o file3.o
```

The advantage of using *make* becomes clear if we type `make myprogram` again:

```
make myprogram

/* Output */
make: ‘myprogram’ is up to date.
```

The *make* program sees that `myprogram` already exists and that there is no reason to compile it again. However, if we change `file2.c` and type `make myprogram`:

```
make myprogram

/* Output */
gcc -Wall -c file2.c
gcc -o myprogram file1.o file2.o file3.o
```

The *make* program sees that `file2.c` has changed. Therefore, it first recompiles this file and the links a new `myprogram`.

To find more information about *make*, have a look at the man page (`man make`) or go to `http://www.gnu.org/manual/make/html_mono/make.html`.

4.6 Debugging

Since the C language is less strict that Java, it is easier to make mistakes. In this section, we will briefly look at ways to prevent and find errors.

4.6.1 Compiler flags

The best way of handling errors is to prevent them. The compiler can help by giving you warnings when sees suspicious code. By turning on the compiler options `-Wall`, `-W`, `-Wundef`, `-Wunreachable-code`, `-Wshadow` and `-pedantic`, many frequently made errors can be avoided. A description of these and other compiler options can be found at `http://gcc.gnu.org/onlinedocs/gcc_3.html#SEC11`.

4.6.2 Assert

The assert library lets you insert checks into your program so it ’crashes’ in a controlled way. To use assert, include the file `assert.h`.

```
void assert (int expression);
```

This `assert` function prints an error message to standard output and terminates the program if expression is false (i.e., is equal to zero). This only happens when the macro NDEBUG is undefined.

```
#include <assert.h>

int main(void)
{
    double d = 0.0;

    assert(d != 0.0);
    d = 100.0/d;
}

/* Output */
a.out: example.c:7: main: Assertion ‘d != 0.0’ failed.
Aborted (core dumped)
```

4.6.3 Debugger

If your program crashes, it produces a *core dump*. A core dump is a file (called ’`core`’), which contains a copy of all the memory your program was using when it crashed. Using the debugger program *gdb*, you can inspect your program and the core dump to see what went wrong. Take the following example:

```
int main(void)
{
    int a, b, c;

    a = 0;
    b = 0;
    c = a / b; /* CRASH */
}
```

```
/* compile this program with -g to generate debugging info */
gcc -g x.c

/* run it */
./a.out
Floating point exception (core dumped)

/* run the debugger like this */
gdb a.out core

/* Output */
Copyright 1998 Free Software Foundation, Inc.
...
Program terminated with signal 8, Floating point exception.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
#0  0x80483ce in main () at example.c:7
7               c = a / b;
(gdb)
```

The debugger now tells you that the error occurred at line 7 of `example.c` (which contains '`c = a / b;`') and waits for instructions. Note that we compiled the program using the option '`-g`' to produce extra information for the debugger. More information about *gdb* can be found by typing `man gdb` or at `http://www.gnu.org/manual/gdb/html_chapter/gdb_toc.html`.

Chapter 5

Pointers

Pointers in C are somewhat similar to references in Java. However, there are a lot of differences which make pointers both more powerful and more dangerous to use that references. To understand how a pointer works, we will fist look at variables in a little more detail.

The text in this Chapter was taken from *ed Jensen's Tutorial on Pointers and Arrays in C*, which can be found at `http://home.netcom.com/~tjensen/ptr/cpoint.htm`.

5.1 Variables

A variable in a program is something with a name, the value of which can vary. The way the compiler handles variables, is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary (i.e., the *type* of the variable) For example, an integer variable is 4 bytes, and that a double variable is 8 bytes (note that this depends on the processor architecture and the compiler used). When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name k by writing:

```
int k;
```

On seeing the "int" part of this statement the compiler sets aside 4 bytes of memory to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol k and the relative address (position) in memory where those 4 bytes were set aside. Thus, later if we write:

```
k = 2;
```

we expect that, at run time when this statement is executed, the value 2 will be placed in that memory location reserved for the storage of the value of k. In C we refer to a variable such as the integer k as an ”object” (be careful that object means something different that ’Object’ in Java!).

In a sense there are two ”values” associated with the object `k`. One is the value of the integer stored there (2 in the above example) and the location of `k` in the memory (i.e., the address of k). Some texts refer to these two values as the rvalue (right value, pronounced ”are value”) and lvalue (left value, pronounced ”el value”) respectively.

In some languages, the lvalue is the value permitted on the left side of the assignment operator ’=’ (i.e., the address where the result of evaluation of the right side ends up). The rvalue is that which is on the right side of the assignment statement, the 2 above. Rvalues cannot be used on the left side of the assignment statement. Thus: 2 = k; is illegal.

Actually, the above definition of ”lvalue” is somewhat modified for C. According to K&R II (page 197) [2]: *”An object is a named region of storage; an lvalue is an expression referring to an object.”* However, at this point, the definition originally cited above is sufficient. As we become more familiar with pointers we will go into more detail on this.

Now consider:

```
int j, k;

k = 2;
j = 7;    <-- line 1, copy 7 TO the memory position called ’j’
k = j;    <-- line 2, copy value FROM the memory position
                      called ’j’ TO the to the memory position
                      called ’k’
```

In the above, the compiler interprets the j in line 1 as the address of the variable j (its lvalue) and creates code to copy the value 7 to that address. In line 2, however, the j is interpreted as its rvalue (since it is on the right hand side of the assignment operator ’=’). That is, here the j refers to the value stored at the memory location set aside for j, in this case 7. So, the 7 is copied to the address designated by the lvalue of k.

In all of these examples, we are using 4 byte integers so all copying of rvalues from one storage location to the other is done by copying 4 bytes. Had we been using doubles, we would be copying 8 bytes.

The C language has a special keyword *sizeof*, which allows the programmer to retrieve the size of a type (i.e., the number of bytes required to store a value of that type). This keyword can be used like this:

```
struct example {
    int value1, value2, value3;
};
sizeof(int);            /* result = 4  */
sizeof(struct example); /* result = 12 */
```

5.2 Creating and using pointers

Now, let's say that we have a reason for wanting a variable designed to hold an address (or lvalue). The size required to hold such a value depends on the system. On old computers with 64K of memory total, the address of any point in memory can be contained in 2 bytes. Computers with more memory would require more bytes to hold an address. Some computers, such as the PC might require special handling to hold a segment and offset under certain circumstances. The actual size required is not too important so long as we have a way of informing the compiler that what we want to store is an address.

Such a variable is called a *pointer* variable (for reasons which hopefully will become clearer a little later). In C when we define a pointer variable we do so by preceding its name with an asterisk ('`*`'). In C we also give our pointer a type which refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
int *ptr;
```

ptr is the name of our variable (just as k was the name of our integer variable). The '`*`' informs the compiler that we want a pointer variable (i.e., to set aside however many bytes is required to store an address in memory). The `int` says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer.

If we don't give ptr a value at its declaration, it is best to initialized it to a special *null* value (or *null pointer*. The actual bit pattern used for a null pointer depends on the specific system on which the code is developed, and may or may not be zero. Therefore, to make the source code compatible between various compilers on various systems, a macro is used to represent a null pointer. That macro goes under the name NULL. Thus, setting the value of a pointer using the NULL macro, as with an assignment statement such as `ptr = NULL`, guarantees that the pointer has become a null pointer. Similarly, one can test for a null pointer using `if (ptr == NULL)`.

But, back to using our new variable ptr. Suppose now that we want to store in ptr the address of our integer variable k. To do this we use the unary '`&`' operator and write:

```
    ptr = &k;
```

What the '`&`' operator does is retrieve the address (lvalue) of k, even though k is on the right hand side of the assignment operator '=', and copies that to the contents of our pointer ptr. Now, ptr is said to "point to" k. Bear with us now, there is only one more operator we need to discuss.

The "dereferencing operator" is the asterisk and it is used as follows:

```
   *ptr = 7;
```

will copy 7 to the address pointed to by ptr. Thus if ptr "points to" (contains the address of) k, the above statement will set the value of k to 7. That is, when we use the '*' this way we are referring to the value of that which ptr is pointing to, not the value of the pointer itself. Similarly, we could write:

```
  printf("%d\n",*ptr);
```

to print to the screen the integer value stored at the address pointed to by ptr. One way to see how all this stuff fits together would be to run the following program and then review the code and the output carefully.

```
/* Program 1.1 from PTRTUT10.TXT   6/10/97 */

#include <stdio.h>

int j, k;
int *ptr;

int main(void)
{
    j = 1;
    k = 2;
    ptr = &k;
    printf("j has value %d and is stored at %p\n", j, &j);
    printf("k has value %d and is stored at %p\n", k, &k);
    printf("ptr has value %p and is stored at %p\n", ptr, &ptr);
    printf("value of integer pointed to by ptr is %d\n", *ptr);

    return 0;
}

/* Output */

j has value 1 and is stored at 0x804972c
k has value 2 and is stored at 0x8049734
ptr has value 0x8049734 and is stored at 0x8049730
value of integer pointed to by ptr is 2
```

In C, like in Java, functions always receive a copy of their parameters. Take the following example:

```
#include <stdio.h>

/* WRONG */
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main(void)
{
    int j = 2;
    int k = 4;

    printf("j has value %d, k has value %d\n", j, k);
    swap(j, k);
    printf("j has value %d, k has value %d\n", j, k);
}

/* Output */
j has value 2, k has value 4
j has value 2, k has value 4
```

As you can see, the implementation of the `swap` function is not correct. When `swap` is invoked by `main`, the local variables `j` and `k` are passed as its parameters. Unfortunately, `swap` only receives copies of their values as parameters. Therefore, as the output shows, `swap` only changes the values of the parameters itself. The values of `j` and `k` remain unchanged.

This problem can be solved by creating a `swap` function that receives pointers to the variables it must swap as parameters (instead of receiving the values of the variables). For example:

```
#include <stdio.h>

/* OK */
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int main(void)
{
    int j = 2;
    int k = 4;

    printf("j has value %d, k has value %d\n", j, k);
    swap(&j, &k);
    printf("j has value %d, k has value %d\n", j, k);
    return 0;
}
/* Output */
j has value 2, k has value 4
j has value 4, k has value 2
```

In this example the `swap` function receives the memory locations (i.e., pointers) of the variables it must swap as parameters. Using these pointers, `swap` can retrieve the values stored in the variables (using the `*` operator) and swap them. Note that the `swap` function now expects pointers to `int` variables as parameters, instead of actual `int` values. Therefore, `main` uses the `&` operator to pass their memory locations instead of their values.

5.3 Pointer types and Arrays

Okay, let's move on. Let us consider why we need to specify the type of variable that a pointer points to, as in:

```
     int *ptr;
```

One reason for doing this is so that later, once ptr ”points to” something, if we write:

```
    *ptr = 2;
```

the compiler will know how many bytes to copy into that memory location pointed to by ptr. If ptr was declared as pointing to an integer, 4 bytes would be copied, if a double, 8 bytes would be copied. Similarly for floats and chars the appropriate number will be copied. But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code. For example, let's say that we have a block in memory consisting if ten integers in a row. That is, 40 bytes of memory are set aside to hold 10 integers (Section 5.10 will explain how this can be done).

Now, let's say we point our integer pointer ptr at the first of these integers. Furthermore lets say that integer is located at memory location 100 (decimal). What happens when we write:

```
    ptr + 1;
```

Because the compiler "knows" this is a pointer (i.e., a memory address) and that it points to an integer (its current address, 100, is the address of an integer), it adds 4 to ptr instead of 1, so the pointer "points to" the next integer, at memory location 104. Similarly, were the ptr declared as a pointer to a double, it would add 8. The same goes for other data types such as floats, chars, or even user defined data types such as structures. This is obviously not the same kind of "addition" that we normally think of. In C it is referred to as addition using *pointer arithmetic*, a term which we will come back to later.

Similarly, since ptr++ is both equivalent to ptr + 1, the unary ++ operator increments the address stored in the pointer by the size of the type pointed to (i.e., sizeof(type pointed to)).

Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers.

Consider the following:

```
    int my_array[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to my_array (i.e., using my_array[0] through my_array[5]). But, we could alternatively access them via a pointer as follows:

```
    int *ptr;
    ptr = &my_array[0];       /* point our pointer at the first
                                 integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
/* Program 2.1 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;

    /* point our pointer to the first element of the array */
    ptr = &my_array[0];

    for (i = 0; i < 6; i++)
    {
      printf("my_array[%d] = %d   ", i, my_array[i]);   /*<-- A */
      printf("ptr + %d = %d\n", i, *(ptr + i));         /*<-- B */
    }
    return 0;
}

/* Output */

my_array[0] = 1   ptr + 0 = 1
my_array[1] = 23   ptr + 1 = 23
my_array[2] = 17   ptr + 2 = 17
my_array[3] = 4   ptr + 3 = 4
my_array[4] = -5   ptr + 4 = -5
my_array[5] = 100   ptr + 5 = 100
```

When we compile and run the above program and we see that lines A and B print out the same values. Also observe how we dereferenced our pointer in line B (i.e., we first added i to it and then dereferenced the new pointer). We could also have written:

```
    printf("ptr + %d = %d\n", i, *ptr++);
```

In C, the standard states that wherever we might use &var_name[0] we can replace that with var_name, thus in our code where we wrote:

```
    ptr = &my_array[0];
```

we can write:

```
    ptr = my_array;
```

to achieve the same result.

This leads many texts to state that the name of an array is a pointer. I prefer to think "*the name of the array is the address of first element in the array*". Many beginners (including myself when I was learning) have a tendency to become confused by thinking of it as a pointer. For example, while we can write

```
    ptr = my_array;
```

we cannot write

```
    my_array = ptr;
```

The reason is that while ptr is a variable, my_array is a *constant.* That is, the location at which the first element of my_array will be stored cannot be changed once my_array[] has been declared.

Modify the example program above by changing

```
    ptr = &my_array[0];
```

to

```
    ptr = my_array;
```

and run it again to verify the results are identical.

5.4 Pointers and Strings

The study of strings is useful to further tie in the relationship between pointers and arrays. It also makes it easy to illustrate how some of the standard C string functions can be implemented. Finally it illustrates how and when pointers can and should be passed to functions.

In C, strings are arrays of characters. This is not necessarily true in other languages. In Java, BASIC, Pascal, Fortran and various other languages, a string has its own data type. But in C it does not. In C a string is an array of characters terminated with a binary zero character (written as '\0'). To start off our discussion we will write some code which you would probably never write in an actual program. Consider, for example:

```
    char my_string[40];

    my_string[0] = 'T';
    my_string[1] = 'e';
    my_string[2] = 'd':
    my_string[3] = '\0';
```

While one would never build a string like this, the end result is a string in that it is an array of characters terminated with a nul character. By definition, in C, a string is an array of characters terminated with the nul character. Be aware that "nul" is not the same as "NULL". The nul refers to a zero as defined by the escape sequence '`\0`'. That is it occupies one byte of memory. NULL, on the other hand, is the name of the macro used to initialize null pointers. NULL is #defined in a header file in your C compiler, nul may not be #defined at all.

Since writing the above code would be very time consuming, C permits two alternate ways of achieving the same thing. First, one might write:

```
    char my_string[40] = {'T', 'e', 'd', '\0',};
```

But this also takes more typing than is convenient. So, C permits:

```
    char my_string[40] = "Ted";
```

When the double quotes are used, instead of the single quotes as was done in the previous examples, the nul character ('`\0`') is automatically appended to the end of the string.

In all of the above cases, the same thing happens. The compiler sets aside an contiguous block of memory 40 bytes long to hold characters and initialized it such that the first 4 characters are `Ted\0`.

Now, consider the following program:

```
/* Program 3.1 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

char strA[80] = "A string to be used for demonstration purposes";
char strB[80];

int main(void)
{
    char *pA;     /* a pointer to type character */
    char *pB;     /* another pointer to type character */

    puts(strA);   /* show string A */

    pA = strA;    /* point pA at string A */
    puts(pA);     /* show what pA is pointing to */

    pB = strB;    /* point pB at string B */
    putchar('\n');       /* move down one line on the screen */

    while(*pA != '\0')   /* line A (see text) */
    {
        *pB++ = *pA++;   /* line B (see text) */
    }
    *pB = '\0';          /* line C (see text) */
    puts(strB);          /* show strB on screen */
    return 0;
}
```

In the above we start out by defining two character arrays of 80 characters each. Since these are globally defined, they are initialized to all '\0's first. Then, strA has the first 42 characters initialized to the string in quotes.

Now, moving into the code, we declare two character pointers and show the string on the screen. We then "point" the pointer pA at strA. That is, by means of the assignment statement we copy the address of strA[0] into our variable pA. We now use the puts() function (from `stdio.h`) to show that which is pointed to by pA on the screen. Consider here that the function prototype for puts() is:

```
    int puts(const char *s);
```

For the moment, ignore the const. The parameter passed to puts() is a pointer to (or the address of) the first character of a string. Thus when we write puts(strA) we are passing the address of strA[0]. Similarly, when we write puts(pA); we are passing the same address, since we have set pA = strA;

Given that, follow the code down to the while() statement on line A. Line A states: 'While the character pointed to by pA (i.e., `*pA`) is not a nul character (i.e., the terminating '\0'), execute line B'. Line B states: 'copy the character pointed to by pA to the space pointed to by pB. Then, increment pA so it points to the next character and increment pB so it points to the next space'.

When we have copied the last character, pA now points to the terminating nul character and the loop ends. However, we have not copied the nul character. And, by definition a string in C must be nul terminated. So, we add the nul character with line C.

Getting back to the prototype for puts() for a moment, the "const" used as a parameter modifier informs the user that the function will not modify the string pointed to by s (i.e., it will treat that string as a constant).

Of course, what the above program illustrates is a simple way of copying a string. After playing with the above until you have a good understanding of what is happening, we can proceed to creating our own replacement for the standard `strcpy()` function that comes with C. It might look like:

```
char *my_strcpy(char *destination, char *source)
{
    char *p = destination;
    while (*source != '\0')
    {
        *p++ = *source++;
    }
    *p = '\0';
    return destination;
}
```

In this case, I have followed the practice used in the standard routine of returning a pointer to the destination.

Again, the function is designed to accept the values of two character pointers (i.e., addresses), and thus in the previous program we could write:

```
int main(void)
{
    my_strcpy(strB, strA);
    puts(strB);
}
```

I have deviated slightly from the form used in standard C which would have the prototype:

```
char *my_strcpy(char *destination, const char *source);
```

Here the "const" modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer (this will be checked by the compiler).

Recall again that a string is nothing more than an array of characters, with the last character being a '\0'. What we have done above is deal with copying an array. It happens to be an array of characters but the technique could be applied to an array of integers, doubles, etc. In those cases, however, we would not be dealing with strings and hence the end of the array would not be marked with a special value like the nul character. We could implement a version that relied on a special value to identify the end. For example, we could copy an array of positive integers by marking the end with a negative integer. On the other hand, it is more usual that when we write a function to copy an array of items other than strings we pass the function the number of items to be copied as well as the address of the array, e.g. something like the following prototype might indicate:

```
    void int_copy(int *ptrA, int *ptrB, int len);
```

where len is the number of integers to be copied. You might want to play with this idea and create an array of integers and see if you can write the function int_copy() and make it work.

This permits using functions to manipulate large arrays. For example, if we have an array of 5000 integers that we want to manipulate with a function, we need only pass to that function the address of the array (and any auxiliary information such as len above). The array itself does not get passed, only its address is sent.

This is different from passing, say an integer, to a function. When we pass an integer we make a copy of the integer (i.e., pass a copy of its value). Within the function any manipulation of the value passed can in no way effect the original integer. But, with arrays and pointers we can pass the address of a variable, and manipulate the value that variable directly.

5.5 More on Strings

Well, we have progressed quite a way in a short time! Let's back up a little and look at what was done in the previous Section on copying of strings in a different light. Consider the following function:

```
    char *my_strcpy(char dest[], char source[])
    {
        int i = 0;
        while (source[i] != '\0')
        {
            dest[i] = source[i];
            i++;
        }
        dest[i] = '\0';
        return dest;
    }
```

Recall that strings are arrays of characters. Here we have chosen to use array notation instead of pointer notation to do the actual copying. The results are the same, the string gets copied using this notation just as accurately as it did before. This raises some interesting points which we will discuss.

Since parameters are passed by value, in both the passing of a character pointer or the name of the array as above, what actually gets passed is the address of the first element of each array. Thus, the numerical value of the parameter passed is the same whether we use a character pointer or an array name as a parameter. This would tend to imply that somehow source[i] is the same as *(p+i).

In fact, this is true. Wherever one writes a[i] it can be replaced with *(a + i) without any problems. In fact, the compiler will create the same code in either case. Thus we see that pointer arithmetic is the same thing as array indexing. Either syntax produces the same result.

This is NOT saying that pointers and arrays are the same thing, they are not. We are only saying that to identify a given element of an array we have the choice of two syntaxes, one using array indexing and the other using pointer arithmetic, which yield identical results.

Now, lets look at this last expression. Part of it, (a + i), is a simple addition using the + operator and the rules of C state that such an expression is commutative. That is (a + i) is identical to (i + a). Thus we could write *(i + a) just as easily as *(a + i).

Now, looking at our function above, when we write:

```
    dest[i] = source[i];
```

due to the fact that array indexing and pointer arithmetic yield identical results, we can write this as:

```
    *(dest + i) = *(source + i);
```

But, this takes 2 additions for each value taken on by i. Additions, generally speaking, take more time than incrementations (such as those done using the ++ operator as in i++). This may not be true in modern optimizing compilers, but one can never be sure. Thus, the pointer version may be a bit faster than the array version.

Another way to speed up the pointer version would be to change:

```
    while (*source != '\0')
```

to simply

```
    while (*source)
```

since the value within the parenthesis will go to zero (*false*) at the same time in either case.

At this point you might want to experiment a bit with writing some of your own programs using pointers. Manipulating strings is a good place to experiment. You might want to write your own versions of such standard functions as:

```
    strlen();
    strcat();
    strchr();
    strcpy();
```

and any others you might have on your system. These functions can be imported by including the `string.h` header file. For example:

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

char name[40];

int main(void)
{
    strcpy(name, "Jason"); /* copies the string "Jason" into
                              the array */
    return 0;
}
```

We will come back to strings and their manipulation through pointers in a later section.

5.6 Pointers and Structures

As explained in Section 3.4, we can declare the form of a block of data containing different data types by means of a structure declaration. For example, a personnel file might contain structures which look something like:

```
    struct tag {
        char lname[20];        /* last name */
        char fname[20];        /* first name */
        int age;               /* age */
        float rate;            /* e.g. 12.75 per hour */
    };
```

Let's say we have a bunch of these structures in a disk file and we want to read each one out and print out the first and last name of each one so that we can have a list of the people in our files. The remaining information will not be printed out. We will want to do this printing with a function call and pass to that function a pointer to the structure at hand. For demonstration purposes I will use only one structure for now. But realize the goal is the writing of the function, not the reading of the file which, presumably, we know how to do.

For review, recall that we can access structure members with the dot operator as in:

```
/* Program 5.1 from PTRTUT10.HTM     6/13/97 */
#include <stdio.h>
#include <string.h>

struct tag {
    char lname[20];      /* last name */
    char fname[20];      /* first name */
    int age;             /* age */
    float rate;          /* e.g. 12.75 per hour */
};

struct tag my_struct;       /* declare the structure my_struct */

int main(void)
{
    strcpy(my_struct.lname,"Jensen");
    strcpy(my_struct.fname,"Ted");
    printf("%s ", my_struct.fname);
    printf("%s\n", my_struct.lname);
    return 0;
}
```

If we have a large number of employees, what we want to do is manipulate the data in these structures by means of functions. For example we might want a function print out the name of the employee listed in any structure passed to it.

Consider the case described. We want a function that will accept as a parameter a pointer to a structure and from within that function we want to access members of the structure. For example we want to print out the name of the employee in our example structure.

Okay, so we know that our pointer is going to point to a structure declared using struct tag. We declare such a pointer with the declaration:

```
    struct tag *st_ptr;
```

and we point it to our example structure with:

```
    st_ptr = &my_struct;
```

Now, we can access a given member by de-referencing the pointer. But, how do we de-reference the pointer to a structure? Well, consider the fact that we might want to use the pointer to set the age of the employee. We would write:

```
    (*st_ptr).age = 63;
```

Look at this carefully. It says, replace that within the parenthesis with that which st_ptr points to, which is the structure my_struct. Thus, this breaks down to the same as my_struct.age. However, this is a fairly often used expression and the designers of C have created an alternate syntax with the same meaning which is:

```
    st_ptr->age = 63;
```

We can now add our function to program:

```
/* Program 5.2 from PTRTUT10.HTM   6/13/97 */
#include <stdio.h>
#include <string.h>

struct tag{                     /* the structure type */
    char lname[20];             /* last name */
    char fname[20];             /* first name */
    int age;                    /* age */
    float rate;                 /* e.g. 12.75 per hour */
};

struct tag my_struct;           /* define the structure */

void show_name(struct tag *p)
{
    printf("%s ", p->fname);  /* p points to a structure */
    printf("%s ", p->lname);
    printf("%d\n", p->age);
}

int main(void)
{
    struct tag *st_ptr;       /* a pointer to a structure */
    st_ptr = &my_struct;      /* point the pointer to my_struct */
    strcpy(my_struct.lname,"Jensen");
    strcpy(my_struct.fname,"Ted");
    my_struct.age = 63;
    show_name(st_ptr);          /* pass the pointer */
    return 0;
}
```

The show_name function receives a pointer to our struct, and uses the '->' operator to access the fields of that struct.

5.7 Some more on Strings, and Arrays of Strings

Let's go back to strings. In the following all assignments are to be understood as being global (i.e., made outside of any function).

We pointed out in an earlier section that we could write:

```
char my_string[40] = "Ted";
```

which would allocate space for a 40 byte array and put the string in the first 4 bytes (three for the characters in the quotes and a 4th to handle the terminating '\0').

Actually, if all we wanted to do was store the name "Ted" we could write:

```
char my_name[] = "Ted";
```

and the compiler would count the characters, leave room for the nul character and store the total of the four characters in memory the location of which would be returned by the array name, in this case my_name.

In some code, instead of the above, you might see:

```
char *my_name = "Ted";
```

which is an alternate approach. There is a difference between these two. In the array notation, my_name is just the name of a block of memory (containing the string "Ted"). In other words, it is short for &myname[0] which is the address of the first element of the array, and my_name is a constant (i.e., it can not be changed). Only the content of the my_name array can be changed during run time.

In the pointer notation my_name is a variable, which points to a block of memory containing the string "Ted". As a result, both my_name and the content of the array referenced by my_name can be changed during run time.

As to which is the better method, that depends on what you are going to do within the rest of the program.

Let's now go one step further and consider what happens if each of these declarations are done within a function as opposed to globally outside the bounds of any function.

```
void my_function_A(char *ptr)
{
    char a[] = "ABCDE"
    ...
}


void my_function_B(char *ptr)
{
    char *cp = "FGHIJ"
    ...
}
```

In the case of my_function_A, the content, or value(s), of the array a[] is considered to be the data. The array is said to be initialized to the values ABCDE. In the case of my_function_B, the value of the pointer cp is considered to be the data. The pointer has been initialized to point to the string FGHIJ. In both my_function_A and my_function_B the definitions are local variables and thus the string ABCDE is stored on the stack, as is the value of the pointer cp. The string FGHIJ can be stored anywhere. On my system it gets stored in the data segment.

As long as we are discussing the relationship/differences between pointers and arrays, let's move on to multi-dimensional arrays. Consider, for example the array:

```
    char multi[5][10];
```

Just what does this mean? Let's take '`multi[5]`' to be the "name" of an array. Then prepending the char and appending the [10] we have an array of 10 characters. But, the name '`multi[5]`' is in itself an array indicating that there are 5 elements each being an array of 10 characters. Hence we have an array of 5 arrays of 10 characters each.

Assume we have filled this two dimensional array with data of some kind. In memory, it might look as if it had been formed by initializing 5 separate arrays using something like:

```
    multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
    multi[1] = {'a','b','c','d','e','f','g','h','i','j'}
    multi[2] = {'A','B','C','D','E','F','G','H','I','J'}
    multi[3] = {'9','8','7','6','5','4','3','2','1','0'}
    multi[4] = {'J','I','H','G','F','E','D','C','B','A'}
```

At the same time, individual elements might be addressable using syntax such as:

```
    multi[0][3] = '3'
    multi[1][7] = 'h'
    multi[4][0] = 'J'
```

Since arrays are contiguous in memory, our actual memory block for the above should look like:

```
    0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA
    ^
    |
    +----- starting at the address &multi[0][0]
```

Note that I did not write multi[0] = "0123456789". Had I done so a terminating '\0' would have been implied since whenever double quotes are used a '\0' character is appended to the characters contained within those quotes. Had that been the case I would have had to set aside room for 11 characters per row instead of 10.

My goal in the above is to illustrate how memory is laid out for 2 dimensional arrays. That is, this is a 2 dimensional array of characters, NOT an array of "strings".

Now, the compiler knows how many columns are present in the array so it can interpret multi + 1 as the address of the 'a' in the 2nd row above. That is, it adds 10, the number of columns, to get this location. If we were dealing with integers and an array with the same dimension the compiler would add 10*sizeof(int) which, on my machine, would be 20. Thus, the address of the 9 in the 4th row above would be &multi[3][0] or *(multi + 3) in pointer notation. To get to the content of the 2nd element in the 4th row we add 1 to this address and dereference the result as in

```
    *(*(multi + 3) + 1)
```

With a little thought we can see that:

```
    *(*(multi + row) + col)    and
    multi[row][col]            yield the same results.
```

The following program illustrates this using integer arrays instead of character arrays.

```
/* Program 6.1 from PTRTUT10.HTM   6/13/97*/

#include <stdio.h>
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];

int main(void)
{
    int row, col;
    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            multi[row][col] = row*col;
        }
    }

    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            printf("\n%d  ",multi[row][col]);
            printf("%d ",*(*(multi + row) + col));
        }
    }

    return 0;
}
```

Because of the double de-referencing required in the pointer version, the name of a 2 dimensional array is often said to be equivalent to a pointer to a pointer. With a three dimensional array we would be dealing with an array of arrays of arrays and some might say its name would be equivalent to a pointer to a pointer to a pointer. However, here we have initially set aside the block of memory for the array by defining it using array notation. Hence, we are dealing with a constant, not a variable. That is we are talking about a fixed address not a variable pointer. The dereferencing function used above permits us to access any element in the array of arrays without the need of changing the value of that address (the address of multi[0][0] as given by the symbol multi).

5.8 More on Multi-Dimensional Arrays

In the previous section we noted that given

```
    #define ROWS 5
    #define COLS 10

    int multi[ROWS][COLS];
```

we can access individual elements of the array multi using either:

```
    multi[row][col]
```

or

```
    *(*(multi + row) + col)
```

To understand more fully what is going on, let us replace

```
    *(multi + row)
```

with X as in:

```
    *(X + col)
```

Now, from this we see that X is like a pointer since the expression is de-referenced and we know that col is an integer. Here the arithmetic being used is of a special kind called ”pointer arithmetic” is being used. That means that, since we are talking about an integer array, the address pointed to by (i.e., value of) X + col + 1 must be greater than the address X + col by and amount equal to sizeof(int).

Since we know the memory layout for 2 dimensional arrays, we can determine that in the expression multi + row as used above, multi + row + 1 must increase by value an amount equal to that needed to ”point to” the next row, which in this case would be an amount equal to COLS * sizeof(int).

That says that if the expression *(*(multi + row) + col) is to be evaluated correctly at run time, the compiler must generate code which takes into consideration the value of COLS (i.e., the 2nd dimension). Because of the equivalence of the two forms of expression, this is true whether we are using the pointer expression as here or the array expression multi[row][col].

Thus, to evaluate either expression, a total of 5 values must be known:

1. The address of the first element of the array, which is returned by the expression multi (i.e., the name of the array).

2. The size of the type of the elements of the array, in this case sizeof(int).

3. The 2nd dimension of the array
4. The specific index value for the first dimension, row in this case.
5. The specific index value for the second dimension, col in this case.

Given all of that, consider the problem of designing a function to manipulate the element values of a previously declared multidimensional array. For example, one which would set all the elements of the array multi to the value 1.

```
    void set_value(int m_array[][], /* PROBLEM */
                   int rows, int cols)
    {
        int row, col;
        for (row = 0; row < rows; row++)
        {
            for (col = 0; col < cols; col++)
            {
                m_array[row][col] = 1;
            }
        }
    }
```

There is a problem with the parameter of this function. If we try to compile it, the compiler gives the following error:

```
example.c: In function ‘set_value’:
example.c:8: arithmetic on pointer to an incomplete type
```

To calculate the address of the array element ’`m_array[row][col]`, the compiler needs to know the size of the 2nd dimension of the array. Unfortunately, there is no way for the compiler to determine how large this dimension is when the array is passed as a parameter.

The only solution to this problem is to specify (in the parameter list) what the 2nd dimension of the array is going to be:

```
    void set_value(int m_array[][COLS], int rows)
    {
        int row, col;
        for (row = 0; row < rows; row++)
        {
            for (col = 0; col < COLS; col++)
            {
                m_array[row][col] = 1;
            }
        }
    }
```

In fact, in general all dimensions of higher order than one are needed when dealing with multi-dimensional arrays. That is if we are talking about 3 dimensional arrays, the 2nd and 3rd dimension must be specified in the parameter definition.

5.9 Pointers to Arrays

Pointers, of course, can be "pointed at" any type of data object, including arrays. While that was evident when we discussed program 3.1, it is important to expand on how we do this when it comes to multi-dimensional arrays.

To review, in Section 5.3 we stated that given an array of integers we could point an integer pointer at that array using:

```
    int *ptr;                  /* point our pointer at the first
    ptr = &my_array[0];           integer in our array */
```

As we stated there, the type of the pointer variable must match the type of the first element of the array.

In addition, we can use a pointer as a formal parameter of a function which is designed to manipulate an array. For example:

```
    int array[3] = {'1', '5', '7'};
    void a_func(int *p);
```

Some programmers might prefer to write the function prototype as:

```
   void a_func(int p[]);
```

which would tend to inform others who might use this function that the function is designed to manipulate the elements of an array. Of course, in either case, what actually gets passed is the value of a pointer to the first element of the array, independent of which notation is used in the function prototype or definition. Note that if the array notation is used, there is no need to pass the actual dimension of the array since we are not passing the whole array, only the address to the first element.

We now turn to the problem of the 2 dimensional array. As stated in the last section, C interprets a 2 dimensional array as an array of one dimensional arrays. That being the case, the first element of a 2 dimensional array of integers is a one dimensional array of integers. And a pointer to a two dimensional array of integers must be a pointer to that data type. One way of accomplishing this is through the use of the keyword "typedef". typedef assigns a new name to a specified data type. For example:

```
    typedef unsigned char byte;
```

causes the name byte to mean type unsigned char. Hence

```
    byte b[10];     would be an array of unsigned characters.
```

Note that in the typedef declaration, the word byte has replaced that which would normally be the name of our unsigned char. That is, the rule for using typedef is that the new name for the data type is the name used in the definition of the data type. Thus in:

```
    typedef int Array[10];
```

Array becomes a data type for an array of 10 integers (i.e., Array my_arr; declares my_arr as an array of 10 integers) and Array arr2d[5]; makes arr2d an array of 5 arrays of 10 integers each.

Also note that '`Array *p1d;` makes p1d a pointer to an array of 10 integers. Because *p1d points to the same type as arr2d, assigning the address of the two dimensional array arr2d to p1d, the pointer to a one dimensional array of 10 integers is acceptable (i.e., '`p1d = &arr2d[0];`' or '`p1d = arr2d;`' are both correct).

Since the data type we use for our pointer is an array of 10 integers, incrementing p1d by 1 will change its value by 10*sizeof(int) (you can prove this to yourself by writing and running a simple short program).

Now, while using typedef makes things clearer for the reader and easier on the programmer, it is not really necessary. What we need is a way of declaring a pointer like p1d without the need of the typedef keyword. It turns out that this can be done and that

```
    int (*p1d)[10];
```

is the proper declaration. Variable p1d here is a pointer to an array of 10 integers just as it was under the declaration using the Array type. Note that this is different from

```
    int *p1d[10];
```

which would make p1d the name of an array of 10 pointers to type int.

5.10 Dynamic Allocation of Memory

There are times when it is convenient to allocate memory at run time. The are a number of functions you can use to allocate memory, like `malloc()` and `calloc()` (see `man -S 3 malloc`). Using this approach permits postponing the decision on the size of the memory block need to store an array, for example, until run time. Or it permits using a section of memory for the storage of an array of integers at one point in time, and then when that memory is no longer needed it can be freed up for other uses, such as the storage of an array of structures.

```
    int *iptr;
    iptr = malloc(10 * sizeof(int));

    if (iptr == NULL)
    {
        /* ERROR */
    }
```

When memory is allocated, the allocating function (such as malloc()) returns a pointer. The type of this pointer `void` (i.e., it points to something without a type). This void pointer can be assigned to a pointer variable of any type.

The array dimension can now be determined at run time and is not needed at compile time. That is, the 10 above could be a variable read in from a data file or keyboard, or calculated based on some need, at run time.

Because of the equivalence between array and pointer notation, once iptr has been assigned as above, one can use the array notation. For example, one could write:

```
    int k;
    for (k = 0; k < 10; k++)
       iptr[k] = 2;
```

to set the values of all elements to 2.

Even with a reasonably good understanding of pointers and arrays, one place the newcomer to C is likely to stumble at first is in the dynamic allocation of multi-dimensional arrays. In general, we would like to be able to access elements of such arrays using array notation, not pointer notation, wherever possible. Depending on the application we may or may not know both dimensions at compile time. This leads to a variety of ways to go about our task.

As we have seen, when dynamically allocating a one dimensional array its dimension can be determined at run time. Now, when using dynamic allocation of higher order arrays, we never need to know the first dimension at compile time. Whether we need to know the higher dimensions depends on how we go about writing the code. Here I will discuss various methods of dynamically allocating room for 2 dimensional arrays of integers.

First we will consider cases where the 2nd dimension is known at compile time.

METHOD 1: One way of dealing with the problem is through the use of the typedef keyword. To allocate a 2 dimensional array of integers recall that the following two notations result in the same object code being generated:

```
    multi[row][col] = 1;     *(*(multi + row) + col) = 1;
```

It is also true that the following two notations generate the same code:

```
    multi[row]            *(multi + row)
```

Since the one on the right must evaluate to a pointer, the array notation on the left must also evaluate to a pointer. In fact multi[0] will return a pointer to the first integer in the first row, multi[1] a pointer to the first integer of the second row, etc. Actually, multi[n] evaluates to a pointer to that array of integers that make up the n-th row of our 2 dimensional array. That is, multi can be thought of as an array of arrays and multi[n] as a pointer to the n-th array of this array of arrays. Consider now:

```
/* Program 9.1 from PTRTUT10.HTM  6/13/97 */

#include <stdio.h>
#include <stdlib.h>

#define COLS 5

typedef int RowArray[COLS];
RowArray *rptr;

int main(void)
{
    int nrows = 10;
    int row, col;
    rptr = malloc(nrows * COLS * sizeof(int));
    for (row = 0; row < nrows; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            rptr[row][col] = 17;
        }
    }

    return 0;
}
```

Using this approach, rptr has all the characteristics of an array name name, (except that rptr is modifiable), and array notation may be used throughout the rest of the program. That also means that if you intend to write a function to modify the array contents, you must use COLS as a part of the formal parameter in that function, just as we did when discussing the passing of two dimensional arrays to a function.

METHOD 2: In the METHOD 1 above, rptr turned out to be a pointer to type "one dimensional array of COLS integers". It turns out that there is syntax which can be used for this type without the need of typedef. If we write:

```
    int (*xptr)[COLS];
```

the variable xptr will have all the same characteristics as the variable rptr in METHOD 1 above, and we need not use the typedef keyword. Here xptr is a pointer to an array of integers and the size of that array is given by the #defined COLS. The parenthesis placement makes the pointer notation predominate, even though the array notation has higher precedence. Had we written

```
    int *xptr[COLS];
```

we would have defined xptr as an array of pointers holding the number of pointers equal to that #defined by COLS. That is not the same thing at all. However, arrays of pointers have their use in the dynamic allocation of two dimensional arrays, as will be seen in the next 2 methods.

METHOD 3: Consider the case where we do not know the number of elements in each row at compile time (i.e., both the number of rows and number of columns must be determined at run time). One way of doing this would be to create an array of pointers to type int and then allocate space for each row and point these pointers at each row (this is the way in which Java handles multidimensional arrays). Consider:

```
/* Program 9.2 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int nrows = 5;   /* Both nrows and ncols could be evaluated */
    int ncols = 10;  /* or read in at run time */
    int row;
    int **rowptr;
    rowptr = malloc(nrows * sizeof(int *));
    if (rowptr == NULL)
    {
        puts("\nFailure to allocate room for row pointers.\n");
        exit(0);
    }

    printf("\n\n\nIndex Pointer(hex) Pointer(dec) Diff.(dec)");

    for (row = 0; row < nrows; row++)
    {
        rowptr[row] = malloc(ncols * sizeof(int));
        if (rowptr[row] == NULL)
        {
            printf("\nFailure to allocate for row[%d]\n",row);
            exit(0);
        }
        printf("\n%d    %p    %d", row, rowptr[row], rowptr[row]);
        if (row > 0)
            printf("      %d",(int)(rowptr[row] - rowptr[row-1]));
    }

    return 0;
}
```

In the above code rowptr is a pointer to pointer to type int. In this case it points to the first element of an array of pointers to type int. Consider the number of calls to malloc():

```
    To get the array of pointers             1     call
    To get space for the rows                5     calls
                                           -----
                     Total                   6     calls
```

If you choose to use this approach note that while you can use the array notation to access individual elements of the array (e.g., 'rowptr[row][col] = 17;'), it does not mean that the data in the "two dimensional array" is contiguous in memory.

You can, however, use the array notation just as if it were a continuous block of memory. For example, you can write:

```
   rowptr[row][col] = 176;
```

just as if rowptr were the name of a two dimensional array created at compile time. Of course row and col must be within the bounds of the array you have created, just as with an array created at compile time. More information on creating multidimensional arrays can be found at `http://home.netcom.com/~tjensen/ptr/cpoint.htm`.

5.11 Pointers to Functions

Up to this point we have been discussing pointers to data objects. C also permits the declaration of pointers to functions. Pointers to functions have a variety of uses and some of them will be discussed here.

Consider the following real problem. You want to write a function that is capable of sorting virtually any collection of data that can be stored in an array. This might be an array of strings, or integers, or floats, or even structures. The sorting algorithm can be the same for all. For example, it could be a simple bubble sort algorithm, or the more complex shell or quick sort algorithm. We'll use a simple bubble sort for demonstration purposes.

Sedgewick [5] has described the bubble sort using C code by setting up a function which when passed a pointer to the array would sort it. If we call that function bubble(), a sort program is described by bubble_1.c, which follows:

```
/* Program bubble_1.c from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);

int main(void)
{
    int i;
    putchar('\n');
    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

```
void bubble(int a[], int N)
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (a[j-1] > a[j])
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}
```

The bubble sort is one of the simpler sorts. The algorithm scans the array from the second to the last element comparing each element with the one which precedes it. If the one that precedes it is larger than the current element, the two are swapped so the larger one is closer to the end of the array. On the first pass, this results in the largest element ending up at the end of the array. The array is now limited to all elements except the last and the process repeated. This puts the next largest element at a point preceding the largest element. The process is repeated for a number of times equal to the number of elements minus 1. The end result is a sorted array.

Here our function is designed to sort an array of integers. Thus in line 1 we are comparing integers and in lines 2 through 4 we are using temporary integer storage to store integers. What we want to do now is see if we can convert this code so we can use any data type (i.e., not be restricted to integers).

At the same time we don't want to have to analyze our algorithm and the code associated with it each time we use it. We start by removing the comparison from within the function bubble() so as to make it relatively easy to modify the comparison function without having to re-write portions related to the actual algorithm. This results in bubble_2.c:

```
/* Program bubble_2.c from PTRTUT10.HTM   6/13/97 */

   /* Separating the comparison function */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);
int compare(int m, int n);

int main(void)
{
    int i;
    putchar('\n');
    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int a[], int N)

{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare(a[j-1], a[j]))
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}

int compare(int m, int n)
{
    return (m > n);
}
```

If our goal is to make our sort routine data type independent, one way of doing this is to use pointers to type void to point to the data instead of using the integer data type. As a start in that direction let's modify a few things in the above so that pointers can be used. To begin with, we'll stick with pointers to type integer.

```
/* Program bubble_3.c from PTRTUT10.HTM    6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(int *m, int *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

int compare(int *m, int *n)
{
    return (*m > *n);
}
```

```
void bubble(int *p, int N)
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare(&p[j-1], &p[j]))
            {
                t = p[j-1];
                p[j-1] = p[j];
                p[j] = t;
            }
        }
    }
}
```

Note the changes. We are now passing a pointer to an integer (or array of integers) to bubble(). And from within bubble we are passing pointers to the elements of the array that we want to compare to our comparison function. And, of course we are dereferencing these pointer in our compare() function in order to make the actual comparison. Our next step will be to convert the pointers in bubble() to pointers to type void so that that function will become more type insensitive. This is shown in bubble_4.

```
/* Program bubble_4.c from PTRTUT10,HTM   6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(void *m, void *n);

int compare(void *m, void *n)
{
    int *m1, *n1;
    m1 = (int *)m;
    n1 = (int *)n;
    return (*m1 > *n1);
}
```

```
int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int *p, int N)
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare((void *)&p[j-1], (void *)&p[j]))
            {
                t = p[j-1];
                p[j-1] = p[j];
                p[j] = t;
            }
        }
    }
}
```

Note that, in doing this, in compare() we had to introduce the casting of the void pointer types passed to the actual type being sorted. But, as we'll see later that's okay. And since what is being passed to bubble() is still a pointer to an array of integers, we had to cast these pointers to void pointers when we passed them as parameters in our call to compare().

We now address the problem of what we pass to bubble(). We want to make the first parameter of that function a void pointer also. But, that means that within bubble() we need to do something about the variable t, which is currently an integer. Also, where we use t = p[j-1]; the type of p[j-1] needs to be known in order to know how many bytes to copy to the variable t (or whatever we replace t with).

Currently, in bubble_4.c, knowledge within bubble() as to the type of the data being sorted (and hence the size of each individual element) is obtained from the fact that the first parameter is a pointer to type integer. If we are going to be able to use bubble() to sort any type of data, we need to make that pointer a pointer to type void. But, in doing so we are going to lose information concerning the size of individual elements within the array. So, in bubble_5.c we will add a separate parameter to handle this size information.

These changes, from bubble_4.c to bubble_5.c are, perhaps, a bit more extensive than those we have made in the past. So, compare the two modules carefully for differences.

```
/* Program bubble_5.c from PTRTUT10.HTM    6/13/97 */

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

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(void *p, size_t width, int N);
int compare(void *m, void *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr, sizeof(long), 10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%ld ", arr[i]);
    }

    return 0;
}
```

```
void bubble(void *p, size_t width, int N)
{
    int i, j;
    unsigned char buf[4];
    unsigned char *bp = p;

    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare((void *)(bp + width*(j-1)),
                        (void *)(bp + j*width)))  /* 1 */
            {
/*              t = p[j-1];   */
                memcpy(buf, bp + width*(j-1), width);
/*              p[j-1] = p[j];   */
                memcpy(bp + width*(j-1), bp + j*width , width);
/*              p[j] = t;   */
                memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare(void *m, void *n)
{
    long *m1, *n1;
    m1 = (long *)m;
    n1 = (long *)n;
    return (*m1 > *n1);
}
```

Note that I have changed the data type of the array from int to long to illustrate the changes needed in the compare() function. Within bubble() I've done away with the variable t (which we would have had to change from type int to type long). I have added a buffer of size 4 unsigned characters, which is the size needed to hold a long (this will change again in future modifications to this code). The unsigned character pointer *bp is used to point to the base of the array to be sorted, i.e. to the first element of that array.

We also had to modify what we passed to compare(), and how we do the swapping of elements that the comparison indicates need swapping. Use of memcpy() and pointer notation instead of array notation work towards this reduction in type sensitivity.

Again, making a careful comparison of bubble5.c with bubble4.c can result in improved understanding of what is happening and why.

We move now to bubble_6.c where we use the same function bubble() that we used in bubble_5.c to sort strings instead of long integers. Of course we have to change the comparison function since the means by which strings are compared is different from that by which long integers are compared. And,in bubble6.c we have deleted the lines within bubble() that were commented out in bubble_5.c.

```
/* Program bubble_6.c from PTRTUT10.HTM   6/13/97 */

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

#define MAX_BUF 256

char arr2[5][20] = {  "Mickey Mouse",
                      "Donald Duck",
                      "Minnie Mouse",
                      "Goofy",
                      "Ted Jensen" };

void bubble(void *p, int width, int N);
int compare(void *m, void *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 5; i++)
    {
        printf("%s\n", arr2[i]);
    }
    bubble(arr2, 20, 5);
    putchar('\n\n');

    for (i = 0; i < 5; i++)
    {
        printf("%s\n", arr2[i]);
    }
    return 0;
}
```

```
void bubble(void *p, int width, int N)
{
    int i, j, k;
    unsigned char buf[MAX_BUF];
    unsigned char *bp = p;

    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
          k = compare((void *)(bp + width*(j-1)),
                      (void *)(bp + j*width));
          if (k > 0)
            {
             memcpy(buf, bp + width*(j-1), width);
             memcpy(bp + width*(j-1), bp + j*width , width);
             memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare(void *m, void *n)
{
    char *m1 = m;
    char *n1 = n;
    return (strcmp(m1,n1));
}
```

But, the fact that bubble() was unchanged from that used in bubble_5.c indicates that that function is capable of sorting a wide variety of data types. What is left to do is to pass to bubble() the name of the comparison function we want to use so that it can be truly universal. Just as the name of an array is the address of the first element of the array in the data segment, the name of a function decays into the address of that function in the code segment. Thus we need to use a pointer to a function. In this case the comparison function.

Pointers to functions must match the functions pointed to in the number and types of the parameters and the type of the return value. In our case, we declare our function pointer as:

```
   int (*fptr)(const void *p1, const void *p2);
```

Note that were we to write:

```
    int *fptr(const void *p1, const void *p2);
```

we would have a function prototype for a function which returned a pointer to type int. That is because in C the parenthesis () operator have a higher precedence than the pointer * operator. By putting the parenthesis around the string (*fptr) we indicate that we are declaring a function pointer.

We now modify our declaration of bubble() by adding, as its 4th parameter, a function pointer of the proper type. It's function prototype becomes:

```
    void bubble(void *p, int width, int N,
                int(*fptr)(const void *, const void *));
```

When we call the bubble(), we insert the name of the comparison function that we want to use. bubble_7.c illustrate how this approach permits the use of the same bubble() function for sorting different types of data.

```
/* Program bubble_7.c from PTRTUT10.HTM  6/10/97 */

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

#define MAX_BUF 256

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};
char arr2[5][20] = {  "Mickey Mouse",
                      "Donald Duck",
                      "Minnie Mouse",
                      "Goofy",
                      "Ted Jensen" };

void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *));
int compare_string(const void *m, const void *n);
int compare_long(const void *m, const void *n);
```

```
void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *))
{
    int i, j, k;
    unsigned char buf[MAX_BUF];
    unsigned char *bp = p;

    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            k = fptr((void *)(bp + width*(j-1)),
             (void *)(bp + j*width));
            if (k > 0)
            {
                memcpy(buf, bp + width*(j-1), width);
                memcpy(bp + width*(j-1), bp + j*width , width);
                memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare_long(const void *m, const void *n)
{
    long *m1, *n1;
    m1 = (long *)m;
    n1 = (long *)n;
    return (*m1 > *n1);
}

int compare_string(const void *m, const void *n)
{
    char *m1 = (char *)m;
    char *n1 = (char *)n;
    return (strcmp(m1,n1));
}
```

```c
int main(void)
{
    int i;
    puts("\nBefore Sorting:\n");

    for (i = 0; i < 10; i++)            /* show the long ints */
    {
        printf("%ld ",arr[i]);
    }
    puts("\n");

    for (i = 0; i < 5; i++)             /* show the strings */
    {
        printf("%s\n", arr2[i]);
    }
    bubble(arr, 4, 10, compare_long);      /* sort the longs */
    bubble(arr2, 20, 5, compare_string);   /* sort the strings */
    puts("\n\nAfter Sorting:\n");

    for (i = 0; i < 10; i++)          /* show the sorted longs */
    {
        printf("%d ",arr[i]);
    }
    puts("\n");

    for (i = 0; i < 5; i++)          /* show the sorted strings */
    {
        printf("%s\n", arr2[i]);
    }
    return 0;
}
```

Bibliography

[1] S. C. Johnson and B. W. Kernighan. The Programming Language B. Technical Report Comp. Sci. Tech. Report, #8, AT&T Bell Laboratories, Januari 1973.

[2] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988. ISBN 0131103628.

[3] M. Richards and C. Whitbey-Strevens. *BCPL: The Language and its Compiler*. Cambridge Univ. Press, 1979.

[4] D. M. Ritchie. The Development of the C Language. April 1993. presented at Second History of Programming Languages conference, Cambridge, Mass., online at http://www.digital.com/info/DTJP03/DTJP03HM.HTM.

[5] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1998. ISBN 0201350882.