# Programming in C[1]

Bharat Kinariwala
University of Hawai'i

Tep Dobry
University of Hawai'i

January 5, 1993

[1]Copyright ©1993 by B. Kinariwala and T. Dobry. All rights reserved.

Petitioner has added a lower case letter to page a. Otherwise, it leaves the original page numbering.

MICROSOFT CORP.
EXHIBIT 1013

# Contents

**Table of Contents** i

**List of Figures** vii

**List of Tables** xiii

**Preface** xvii

**1 Introduction** 1

1.1 Computer System Organization . . . . . 2

1.1.1 Computer Hardware . . . . . 2

1.1.2 Computer Software – The Operating System . . . . . 3

1.1.3 Utility Programs . . . . . 4

1.1.4 User Programs and Applications . . . . . 5

1.2 Representing Data and Program Internally . . . . . 5

1.2.1 Representing Data . . . . . 6

1.2.2 Main Memory . . . . . 15

1.2.3 Representing Programs . . . . . 16

1.3 Designing Programs and the C Language . . . . . 19

1.3.1 Designing The Algorithm . . . . . 19

1.3.2 The C Language . . . . . 23

1.4 Summary . . . . . 24

1.5 References . . . . . 25

1.6 Exercises . . . . . 26

1.7 Problems . . . . . 28

**2 Basic Concepts** **29**

2.1 A Simple C Program . . . . . 30

2.1.1 Developing the Algorithm . . . . . 30

2.1.2 Translating the Algorithm to C . . . . . 31

2.1.3 Running the Program . . . . . 31

2.2 Organization of C Programs — Simple Statements . . . . . 33

2.2.1 Comment Statements . . . . . 33

2.2.2 Defining a Function — `main()` . . . . . 34

2.2.3 Variable Declarations . . . . . 35

2.2.4 The Assignment Statement . . . . . 37

2.2.5 Generating Output . . . . . 40

2.3 Testing the Program . . . . . 42

2.3.1 Debugging the Program . . . . . 43

2.3.2 Documenting the Code . . . . . 47

2.4 Input: Reading Data . . . . . 47

2.5 More C Statements . . . . . 51

2.5.1 Making Decisions with Branches . . . . . 51

2.5.2 Simple Compiler Directives . . . . . 55

2.5.3 More on Expressions . . . . . 57

2.5.4 A Simple Loop — `while` . . . . . 64

2.5.5 Controlling Loop Termination . . . . . 70

2.5.6 More Complex Loop Constructs — Nested Loops . . . . . 74

2.6 Common Errors . . . . . 78

2.7 Summary . . . . . 81

2.8 Exercises . . . . . 83

2.9 Problems . . . . . 86

**3 Designing Programs Top Down 89**

3.1 Designing the Algorithm with Functions . . . . . 90

3.1.1 Implementing the Program with Functions . . . . . 91

3.2 Defining Functions . . . . . 94

3.2.1 Passing Data to and from Functions . . . . . 96

3.2.2 Call by Value and Local Variables . . . . . 99

3.3 Coding Programs for Readability . . . . . 105

3.3.1 The C Preprocessor . . . . . 105

3.3.2 Macros . . . . . 105

3.3.3 Including Header Files . . . . . 113

3.3.4 Conditional Compilation . . . . . 116

3.4 Interacting with the Operating System . . . . . 119

3.4.1 Standard Files and EOF . . . . . 119

3.4.2 Standard Files and Redirection . . . . . 125

3.5 Debugging Guidelines . . . . . 126

3.6 Common Errors . . . . . 127

3.7 Summary . . . . . 128

3.8 Exercises . . . . . 130

3.9 Problems . . . . . 132

**4 Processing Character Data 135**

4.1 A New Data Type: `char` . . . . . 135

4.1.1 The ASCII Character Set . . . . . 138

4.1.2 Operations on Characters . . . . . 140
4.1.3 Character I/O Using `getchar()` and `putchar()` . . . . . 147
4.1.4 Strings vs Characters . . . . . 150
4.2 Sample Character Processing Functions . . . . . 151
4.2.1 Converting Letter Characters . . . . . 152
4.2.2 Converting Digit Characters to Numbers . . . . . 154
4.2.3 Counting Words . . . . . 162
4.2.4 Extracting Words . . . . . 169
4.3 New Control Constructs . . . . . 172
4.3.1 The `switch` Statement . . . . . 172
4.3.2 The `break` Statement . . . . . 178
4.3.3 The `continue` Statement . . . . . 182
4.4 Mixing Character and Numeric Input . . . . . 185
4.5 Menu Driven Programs . . . . . 194
4.6 Common Errors . . . . . 196
4.7 Summary . . . . . 199
4.8 Exercises . . . . . 201
4.9 Problems . . . . . 203

**5 Numeric Data Types and Expression Evaluation 207**

5.1 Representing Numbers . . . . . 207
5.1.1 Signed and Unsigned Integer Types . . . . . 209
5.1.2 Single and Double Precision Floating Point Numbers . . . . . 211
5.2 New Control Constructs . . . . . 212
5.2.1 The `for` Statement . . . . . 212
5.2.2 The `do...while` Statement . . . . . 216
5.3 Scalar Data Types . . . . . 224

5.3.1 Data Type `void` . . . . . 224
5.3.2 Enumeration . . . . . 226
5.3.3 Defining User Types: `typedef` . . . . . 228
5.4 Operators and Expression Evaluation . . . . . 229
5.4.1 Precedence and Associativity . . . . . 232
5.4.2 The Data Type of the Result . . . . . 234
5.4.3 Some New Operators . . . . . 236
5.5 Common Errors . . . . . 244
5.6 Summary . . . . . 246
5.7 Exercises . . . . . 248
5.8 Problems . . . . . 251

**6 Pointers** **255**

6.1 What is a Pointer? . . . . . 255
6.1.1 Data vs Address . . . . . 256
6.1.2 Indirect Access of Values . . . . . 259
6.2 Passing Pointers to Functions . . . . . 268
6.2.1 Indirectly Incrementing a Variable . . . . . 268
6.2.2 Computing the Square and Cube . . . . . 268
6.2.3 A function to Swap Values . . . . . 276
6.3 Returning to the Payroll Task with Pointers . . . . . 276
6.4 Common Errors . . . . . 287
6.5 Summary . . . . . 288
6.6 Exercises . . . . . 290
6.7 Problems . . . . . 292

**7 Arrays** **293**

7.1 A Compound Data Type — *array* . . . . . 294

7.1.1 Declaring Arrays . . . . . 294

7.1.2 Character Strings as Arrays . . . . . 298

7.2 Passing Arrays to Functions . . . . . 302

7.3 Arrays, Pointers, Pointer Arithmetic . . . . . 306

7.3.1 Pointers: Increment and Decrement . . . . . 311

7.3.2 Array Names vs Pointer Variables . . . . . 315

7.4 String Assignment and I/O . . . . . 319

7.5 Array Initializers . . . . . 321

7.6 Arrays for Databases . . . . . 323

7.7 Common Errors . . . . . 327

7.8 Summary . . . . . 329

7.9 Exercises . . . . . 332

7.10 Problems . . . . . 336

**8 Functions and Files** **339**

**9 Sorting and Searching** **341**

**10 String Processing** **343**

**11 Two Dimensional Arrays** **345**

**12 Structures and Unions** **347**

**13 File Input/Output** **349**

**14 Storage Class and Scope** **351**

**15 Engineering Programming Examples** **353**

**A C Language Reference** **355**

**B ANSI C vs Old C** **357**

**C The C Standard Library** **359**

# List of Figures


1.1 Computer System Block Diagram . . . . . 2

1.2 Memory and Addresses . . . . . 16

1.3 CPU and Memory Configuration . . . . . 17

1.4 Machine and Assembly Language Program Fragment . . . . . 19

1.5 Structural Diagram for Payroll Task . . . . . 20

1.6 Flow Chart for Payroll Task . . . . . 21

2.1 Code for pay0.c . . . . . 32

2.2 Allocation of Memory Cells or Objects . . . . . 36

2.3 Assignment of Values . . . . . 37

2.4 Computation of `pay` . . . . . 38

2.5 Program Trace for pay0.c . . . . . 46

2.6 Keyboard Buffer . . . . . 49

2.7 Code for pay1.c . . . . . 50

2.8 If statement control flow . . . . . 53

2.9 Code for pay2.c . . . . . 59

2.10 Control Flow for `while` statement . . . . . 66

2.11 Coding a While Loop . . . . . 67

2.12 Code for pay3.c . . . . . 70

2.13 Code for pay4.c . . . . . 73

2.14 Code for prime.c . . . . . 77

3.1 Structural Diagram for Payroll Task . . . . . 90
3.2 Code for pay5.c driver . . . . . 93
3.3 Function Call to `calc_pay()` . . . . . 94
3.4 Code for `calc_pay()` . . . . . 95
3.5 Function Call Trace . . . . . 97
3.6 Trace for `calc_pay()` . . . . . 99
3.7 Call by value variable allocation . . . . . 101
3.8 Local Variables in Blocks . . . . . 103
3.9 Driver for `niceday.c` . . . . . 111
3.10 Functions for `niceday.c` . . . . . 112
3.11 Using Directives in `niceday.c` . . . . . 121
3.12 End of File Marker . . . . . 122
3.13 Code for maxabs.c . . . . . 124

4.1 Code for copy0.c . . . . . 137
4.2 Code for ASCII Attributes . . . . . 145
4.3 Printing character representations . . . . . 146
4.4 Alternate code for attributes program . . . . . 148
4.5 Using `getchar()` and `putchar()` . . . . . 150
4.6 Strings . . . . . 151
4.7 Code for upper case . . . . . 153
4.8 Code for Character Utilities . . . . . 155
4.9 Header file for Character Utilities . . . . . 156
4.10 Code for `getint()` . . . . . 158
4.11 Revised Character Utility Header File . . . . . 163
4.12 Revised Character Utility Code . . . . . 164
4.13 Code for Count Words Driver . . . . . 166

4.14 Header Files for Word Count . . . . . 167

4.15 Code for Word Count Utility Functions . . . . . 168

4.16 Code fore extracting words . . . . . 171

4.17 Control Flow for switch statement . . . . . 174

4.18 Code for `vowelp()` Using a `switch` Statement . . . . . 175

4.19 Code for `encrypt.c` . . . . . 176

4.20 Implementing `print_next()` Using a `switch` Statement . . . . . 178

4.21 New Implementation of `print_category` using `switch` . . . . . 179

4.22 Extracting Words Using `break` . . . . . 181

4.23 Code for Revised `encrypt.c` . . . . . 184

4.24 Code for Testing `scanf()` . . . . . 185

4.25 Revised Code for Reading Integers . . . . . 188

4.26 Mixing Numeric and Character Data . . . . . 190

4.27 Revised Code for Mixing Data . . . . . 192

4.28 A Better Revision for Mixing Data . . . . . 193

4.29 Code for menu driven program . . . . . 195

4.30 Menu Driven Functions . . . . . 197

5.1 Control Flow of `for` Loop . . . . . 213

5.2 Code for factorial . . . . . 214

5.3 Control Flow of `do...while` Loop . . . . . 216

5.4 Code for Square Root . . . . . 219

5.5 Code for Math Utilities . . . . . 220

5.6 Modified Square Root Driver . . . . . 222

5.7 Modified Square Root Utilities . . . . . 223

5.8 Code for Simple Postfix Calculator . . . . . 230

5.9 Code for `get_operator()` . . . . . 231

5.10 Factorial Function Using Composite Operators . . . . . 239

5.11 Function `maxdbl` Using a Conditional Expression . . . . . 240

5.12 Revised Fibonacci . . . . . 241

5.13 Testing `sizeof` Operator . . . . . 244

6.1 Declaring Pointer Variables . . . . . 258

6.2 Declaration of Pointer Variables . . . . . 260

6.3 Assignments of pointers . . . . . 260

6.4 Effect of Pointer to Pointer Assignment . . . . . 261

6.5 Effect of Pointer Reassignment . . . . . 262

6.6 Effect of Indirect Pointer Access and Assignment . . . . . 263

6.7 Effect of Indirect Assignment . . . . . 263

6.8 Effect of Indirect Pointer Access and Assignment . . . . . 264

6.9 Example Code with Direct and Indirect Access . . . . . 266

6.10 Trace for Direct and Indirect Access . . . . . 267

6.11 Code for Indirect Access by a Function . . . . . 269

6.12 Trace for Indirect Access by a Function . . . . . 270

6.13 Code for Indirectly Returned Values . . . . . 271

6.14 Trace for `sqcube` . . . . . 272

6.15 Trace for `sqcube` . . . . . 273

6.16 Trace for `sqcube` . . . . . 274

6.17 Trace for `sqcube` . . . . . 275

6.18 Code for a Function, `swap()` . . . . . 277

6.19 Trace for `swap()` . . . . . 278

6.20 Trace for `swap()` . . . . . 279

6.21 Trace for `swap()` . . . . . 280

6.22 Trace for `swap()` . . . . . 281

6.23 Header file `payutil.h` . . . . . 283

6.24 Code for the Driver for `pay6.c` . . . . . 284

6.25 Code for `print_data()` and `print_pay()` . . . . . 285

6.26 Code for `calc_pay()` . . . . . 286

6.27 Code for `get_data()` . . . . . 286

7.1 Code for `scores.c` . . . . . 295

7.2 An Array of size MAX . . . . . 296

7.3 A String Stored in Memory . . . . . 299

7.4 Code for `string.c` . . . . . 300

7.5 Code fore `scores.c` . . . . . 304

7.6 Code for `string2.c` . . . . . 305

7.7 Pointer Arithmetic . . . . . 307

7.8 Array Pointers as Function Parameters . . . . . 309

7.9 Pointer to a Sub-array . . . . . 312

7.10 Pointer Variables and Arrays . . . . . 317

7.11 Pointer Variables and Strings . . . . . 318

7.12 Data Record Spread Over Several Arrays . . . . . 323

7.13 Code for `paytab.c` . . . . . 324

7.14 Code for `payutil.c` . . . . . 325

# List of Tables


1.1 Number Representations . . . . . 14

2.1 Arithmetic Operators . . . . . 40

2.2 Relational Operators . . . . . 55

2.3 Precedence and Associativity of Operators . . . . . 59

2.4 Logical Operator Symbols in C . . . . . 62

2.5 Truth Table for Logical Combinations . . . . . 63

4.1 Escape Sequences . . . . . 136

4.2 ASCII Table . . . . . 139

4.3 Escape sequences with Octal & Hexadecimal values . . . . . 141

5.1 Precedence and Associativity Table . . . . . 233

5.2 Composite Assignment Operators and Their Equivalents . . . . . 238

5.3 Space allocation in Bytes for data types . . . . . 243

7.1 Pointer Arithmetic and Indirect Access . . . . . 307

# Preface

The C language has boomed in popularity and availability since its creation in the 70's. It has largely become the language of choice for systems programming as well as general purpose programming in both the numeric and symbolic realms. As a result, all programmers today should have some working knowledge of C, particularly in engineering.

This book is intended to be a first text in programming in general with emphasis on the C language. It is meant for students with little or no previous programming experience and as such, a primary focus is on the top down *design* of programs, beginning with the development of an algorithm, proceeding to the translation of the algorithm into a programming language (C), and the subsequent testing and debugging of the resulting code. Throughout the text, emphasis is placed on organization and readability of code as well as debugging aids in program development. In addition, understandable and functional user interfaces are described.

As an introductory text on programming, our approach is to motivate the introduction of features of the language through example problems. We start with meaningful but simple tasks, develop an algorithm to solve the task and then introduce the necessary language constructs to implement the algorithm. We then refine the task, adding complexity or desirable features to motivate introduction of new language constructs. As such, the text is not meant to be a C reference manual, but a text on program design utilizing the available language features to implement the design. However, for student's reference, the key constructs introduced are summarized at the end of each chapter. The intent is for the student to be able to design and code programs from the very beginning.

The book is organized as follows:

- Chapter 1 is an introduction to computers and some of the terminology used throughout the text.
- Chapter 2 begins the development of a simple C program and the introduction to the organization and basic statements of the language.
- Chapter 3 stresses the top down approach to design and introduces functions at an early stage to emphasize their relation to algorithms.
- Chapter 4 introduces the character data type and algorithms for processing characters.

- Chapter 5 presents numeric data types and their limitations and discusses the details of C expression evaluation.
- Chapter 6 addresses the important concept of pointers and their use in C in functions.
- Chapter 7 introduces compound data types with arrays and discusses their relation with pointers.
- Chapter 8 describes some of the standard library functions provided in C for character and math processing as well as giving a detailed description of the standard I/O functions printf() and scanf() and their variations for file I/O.
- Chapter 9 presents some standard sorting and searching techniques.
- Chapter 10 describes the powerful string processing utilities in C and the concept of libraries of functions.
- Chapter 11 returns to arrays presenting two dimensional arrays.
- Chapter 12 discusses the remaining compound data type; structures and unions.
- Chapter 13 presents advanced file Input/Output features of the language.
- Chapter 14 describes the memory organization of C programs and discusses the details of storage classes and scope.
- Finally, Chapter 15 provides several examples of algorithms useful in Engineering computation. It makes use of the concepts presented in earlier Chapters and these examples may be discussed with the appropriate Chapter.

In addition, three Appendices are provided as follows:

- Appendix A provides a summary of the C language constructs discussed in the text.
- Appendix B contrasts the language features of ANSI C as presented in this text to "old" C which is still available on many Unix systems.
- Appendix C summarizes the standard library functions available in C.

# Chapter 7

# Arrays

A programmer is concerned with developing and implementing algorithms for a variety of tasks. As tasks become more complex, algorithm development is facilitated by structuring or organizing data in specialized ways. There is no best data structure for all tasks; suitable data structures must be selected for the specific task. Some data structures are provided by programming languages; others must be derived by the programmer from available data types and structures.

So far we have used integer, floating point and character data types as well as pointers to them. These data types are called **base** or **scalar** data types. Such base data types may be used to derive data structures which are organized groupings of instances of these types. The C language provides some widely used **compound** or **derived** data types together with mechanisms which allow the programmer to define variables of these types and access the data stored within them.

The first such type we will discuss is called an **array**. Many tasks require storing and processing a list of data items. For example, we may need to store a list of exam scores and to process it in numerous ways: find the maximum and minimum, average the scores, sort the scores in descending order, search for a specific score, etc. Data items in simple lists are usually of the same scalar type; for example a list of exam scores consists of all integer type items. We naturally think of a list as a data structure that should be referenced as a unit. C provides a derived data type that stores such a list of objects where each object is of the same data type — the array.

In this chapter, we will discuss arrays; how they are declared and data is accessed in an array. We will discuss the relationship between arrays and pointers and how arrays are passed as arguments in function calls. We will present several example programs using arrays, including a revision of our "payroll" task from previous chapters. One important use of arrays is to hold strings of characters. We will introduce strings in this chapter and show how they are stored in C; however, since strings are important in handling non-numeric data, we will discuss string processing at length in Chapter 10.

## 7.1 A Compound Data Type — *array*

As described above, an array is a compound data type which allows a collection of data of the same type to be grouped into a single object. As with any data type, to understand how to use an array, one must know how such a structure can be declared, how data may be stored and accessed in the structure, and what operations may be performed using this new type.

### 7.1.1 Declaring Arrays

Let us consider the task of reading and printing a list of exam scores.

LIST0: Read and store a list of exam scores and then print it.

Since we are required to store the entire list of scores before printing it, we will use an array hold the data. Successive elements of the list will be stored in successive elements of the array. We will use a counter to indicate the next available position in the array. Such a counter is called an **index** into the array. Here is an algorithm for our task:

```
initialize the index to the beginning of the array
while there are more data items
     read a score and store in array at the current index
     increment index
set another counter, count = index - the number of items in the array
traverse the array: for  each index starting at the beginning to count
     print the array element at index
```

The algorithm reads exam scores and stores them in successive elements of an array. Once the list is stored in an array, the algorithm traverses the array, i.e. accesses successive elements, and prints them. A count of items read in is kept and the traversal continues until that count is reached.

We can implement the above algorithm in a C program as shown in Figure 7.1. Before explaining this code, here is a sample session generated by executing this program:

```
***List of Exam Scores***

Type scores, EOF to quit
67
75
82
69
^D

***Exam Scores***
```

```
/*  File: scores.c
    This program reads a list of integer exam scores and prints them out.
*/
#include <stdio.h>
#define MAX 100

main()
{   int exam_scores[MAX], index, n, count;

    printf("***List of Exam Scores***\n\n");
    printf("Type scores, EOF to quit\n");

    /* read scores and store them in an array */
    index = 0;
    while ((index < MAX) && (scanf("%d", &n) != EOF))
         exam_scores[index++] = n;
    count = index;

    /* print scores from the array */
    printf("\n***Exam Scores***\n\n");
    for (index = 0; index < count; index++)
         printf("%d\n", exam_scores[index]);
}
```

Figure 7.1: Code for `scores.c`

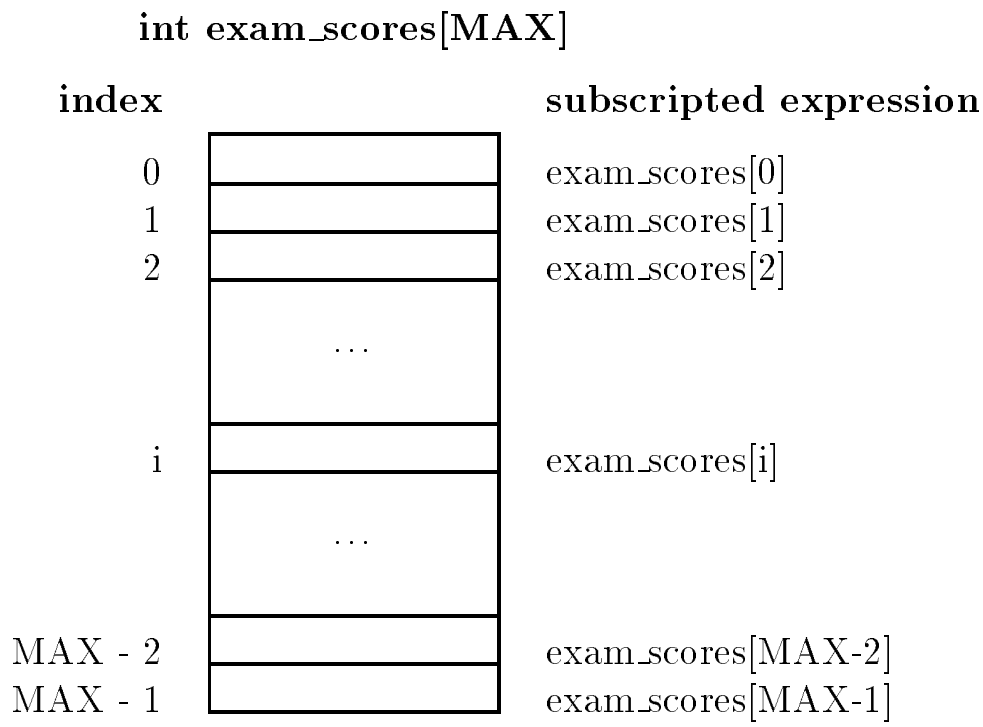


Figure 7.2: An Array of size MAX

```
67
75
82
69
```

Referring to the code in Figure 7.1, the program first declares an array, `exam_scores[MAX]`, of type integer. This declaration allocates a contiguous block of memory for objects of integer type as shown in Figure 7.2. The macro, `MAX`, in square brackets gives the **size** of the array, i.e. the number of elements this compound data structure is to contain. The name of the array, `exam_scores`, refers to the entire collection of `MAX` integer cells. Individual objects in the array may be accessed by specifying the name of the array and the **index**, or element number, of the object; a process called **indexing**. In C, the elements in the array are numbered from 0 to MAX - 1. So, the elements of the array are referred to as `exam_scores[0]`, `exam_scores[1]`, ..., `exam_scores[MAX - 1]`, where the index of each element is placed in square brackets. These index specifiers are sometimes called **subsctipts**, analogous to the mathematical expression $exam_scores_i$a. These *indexed* or *subscripted* array expressions are the *names* of each object in the array and may be used just like any other variable name.

In the code, the while loop reads a score into the variable, `n`, places it in the array by assigning it to `exam_scores[index]`, and increments `index`. The loop is terminated either when `index` reaches `MAX` (indicating a full array) or when `scanf()` returns `EOF`, indicating the end of the data.

We could have also read each data item directly into `exam_scores[index]` by writing `scanf()` as follows:

```
scanf("%d", &exam_scores[index])
```

We choose to separate reading an item and storing it in the array because the use of the increment operator, `++`, for `index` is clearer if reading and storing of data items are separated.

Once the data items are read and stored in the array, a count of items read is stored in the variable `count`. The list is then printed using a `for` loop. The array is traversed from element 0 to element `count` - 1, printing each element in turn.

From the above example, we have seen how we can declare a variable to be of the compound data type, array, how data can be stored in the elements of the array, and subsequently accessed. More formally, the syntax for an array declaration is:

<type-specifier><identifier>[<size>];

where the <type-specifier> may be any scalar or derived data type; and the <size> must evaluate, at compile time, to an unsigned integer. Such a declaration allocates a contiguous block of memory for objects of the specified type. The data type for each object in the block is specified by the <type-specifier>, and the number of objects in the block is given by —sf <size> as seen in Figure 7.2. As stated above, the index values for all arrays in C must start with 0 and end with the highest index, which is one less than the size of the array. The subscripting expression with the syntax:

<identifier>[<expression>]

is the name of one element object and may be used like any other variable name. The subscript, <expression> must evaluate, at run time, to an integer. Examples include:

```
int a[10];
float b[20];
char s[100];
int  i = 0;

a[3] = 13;
a[5] = 8 * a[3];
b[6] = 10.0;
printf("The value of b[6] is %f\n", b[6]);
scanf("%c", &s[7]);
c[i] = c[i+1];
```

Through the remainder of this chapter, we will use the following symbolic constants for many of our examples:

```
/* File: araydef.h */
#define MAX 20
#define SIZE 100
```

In programming with arrays, we frequently need to initialize the elements. Here is a loop that traverses an array and initializes the array elements to zero:

```
int i, ex[MAX];

for (i = 0; i < MAX; i++)
    ex[i] = 0;
```

The loop assigns zero to `ex[i]` until `i` becomes `MAX`, at which point it terminates and the array elements are all initialized to zero. One precaution to programmers using arrays is that C does not check if the index used as a subscript is within the size of the declared array, leaving such checks as the programmer's responsibility. Failure to do so can, and probably will result in catastrophe.

### 7.1.2 Character Strings as Arrays

Our next task is to store and print non-numeric text data, i.e. a sequence of characters which are called **strings**. A string is an list (or string) of characters stored contiguously with a marker to indicate the end of the string. Let us consider the task:

STRING0: Read and store a string of characters and print it out.

Since the characters of a string are stored contiguously, we can easily implement a string by using an array of characters if we keep track of the number of elements stored in the array. However, common operations on strings include breaking them up into parts (called **substrings**), joining them together to create new strings, replacing parts of them with other strings, etc. There must be some way of detecting the size of a current valid string stored in an array of characters.

In C, a string of characters is stored in successive elements of a character array and terminated by the `NULL` character. For example, the string `"Hello"` is stored in a character array, `msg[]`, as follows:

```
char msg[SIZE];

msg[0] = 'H';
msg[1] = 'e';
msg[2] = 'l';
msg[3] = 'l';
msg[4] = 'o';
msg[5] = '\0';
```

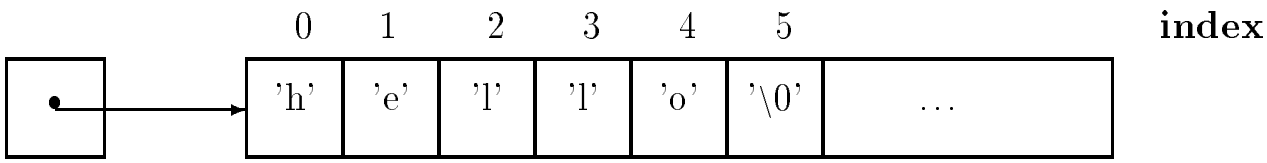


Figure 7.3: A String Stored in Memory

The `NULL` character is written using the escape sequence '\0'. The ASCII value of `NULL` is 0, and `NULL` is defined as a macro to be 0 in `stdio.h`; so programs can use the symbol, `NULL`, in expressions if the header file is included. The remaining elements in the array after the `NULL` may have any garbage values. When the string is retrieved, it will be retrieved starting at index 0 and succeeding characters are obtained by incrementing the index until the first `NULL` character is reached signaling the end of the string. Figure 7.3 shows a string as it is stored in memory.

Given this implementation of strings in C, the algorithm to implement our task is now easily written. We will assume that a string input is a sequence of characters terminated by a newline character. (The newline character is not part of the string). Here is the algorithm:

```
initialize index to zero
while not a newline character
     read and store a character in the array at the next index
     increment the index value
terminate the string of characters in the array with a NULL char.
initialize index to zero
traverse the array until a NULL character is reached
     print the array character at index
     increment the index value
```

The program implementation has:

- a loop to read string characters until a newline is reached;
- a statement to terminate the string with a NULL;
- and a loop to print out the string.

The code is shown in Figure 7.4 and a sample session form the program is shown below.

Sample Session:

```
***Character Strings***

Type characters terminated by a RETURN or ENTER
```

```
/*  File: string.c
    This program reads characters until a newline, stores them in an
    array, and terminates the string with a NULL character. It then prints
    out the string.
*/

#include <stdio.h>
#include "araydef.h"

main()
{   char msg[SIZE], ch;
    int i = 0;

    printf("***Character Strings***\n\n");
    printf("Type characters terminated by a RETURN or ENTER\n");

    while ((ch = getchar()) != '\n')
         msg[i++] = ch;

    msg[i] = '\0';

    i = 0;
    while (msg[i] != '\0')
         putchar(msg[i++]);
    printf("\n");
}
```

Figure 7.4: Code for `string.c`

```
Hello
Hello
```

The first `while` loop reads a character into `ch` and checks if it is a newline, which discarded and the loop terminated. Otherwise, the character is stored in `msg[i]` and the array index, `i`, incremented. When the loop terminates, a `NULL` character is appended to the string of characters. In this program, we have assumed that the size of `msg[]` is large enough to store the string. Since a line on a terminal is 80 characters wide and since we have defined SIZE to be 100, this seems a safe assumption.

The next `while` loop in the program traverses the string and prints each character until a `NULL` character is reached. Note, we do not need to keep a count of the number of characters stored in the array in this program since the first `NULL` character encountered indicates the end of the string. In our program, when the first `NULL` is reached we terminate the string output with a newline.

The assignment expression in the above program:

```
msg[i] = '\0';
```

can also be written as:

```
msg[i] = NULL;
```

or:

```
msg[i] = 0;
```

In the first case, the character whose ASCII value is 0 is assigned to `msg[i]`; where in the other cases, a zero value is assigned to `msg[i]`. The above assignment expressions are identical. The first expression makes it clear that a null character is assigned to `msg[i]`, but the second uses a symbolic constant which is easier to read and understand.

To accommodate the terminating `NULL` character, the size of an array that houses a string must be at least one greater than the expected maximum size of string. Since different strings may be stored in an array at different times, the first `NULL` character in the array delimits a valid strin. The importance of the `NULL` character to signal the end of a valid string is obvious. If there were no `NULL` character inserted after the valid string, the loop traversal would continue to print values interpreted as characters, possibly beyond the array boundary until it fortuitously found a `NULL` (0) character.

The second `while` loop may also be written:

```
while (msg[i] != NULL)
    putchar(msg[i++]);
```

and the `while` condition further simplified as:

```
while (msg[i])
    putchar(msg[i++]);
```

If `msg[i]` is any character with a non-zero ASCII value, the `while` expression evaluates to True. If `msg[i]` is the `NULL` character, its value is zero and thus False. The last form of the `while` condition is the more common usage. While we have used the increment operator in the putchar() argument, it may also be used separately for clarity:

```
while (msg[i]) {
    putchar(msg[i]);
    i++;
}
```

It is possible for a string to be empty; that is, a string may have no characters in it. An empty string is a character array with the `NULL` character in the zeroth index position, `msg[0]`.

## 7.2 Passing Arrays to Functions

We have now seen two examples of the use of arrays — to hold numeric data such as test scores, and to hold character strings. We have also seen two methods for determining how many cells of an array hold useful information — storing a count in a separate variable, and marking the end of the data with a special character. In both cases, the details of array processing can easily obscure the actual logic of a program — processing a set of scores or a character string. It is often best to treat an array as an *abstract data type* with a set of allowed operations on the array which are performed by functional modules. Let us return to our exam score example to read and store scores in an array and then print them, except that we now wish to use functions to read and print the array.

LIST1: Read an array and print a list of scores using functional modules.

The algorithm is very similar to our previous task, except that the details of reading and printing the array is hidden by functions. The function, `read_intaray()`, reads scores and stores them, returning the number of scores read. The function, `print_intaray()`, prints the contents of the array. The refined algorithm for `main()` can be written as:

```
print title, etc.
n = read_intaray(exam_scores, MAX);
print_intaray(exam_scores, n);
```

Notice we have passed an array, `exam_scores`, and a constant, `MAX` (specifying the maximum size of the proposed list), to `read_intarray()` and expect it to return the number of scores placed

in the array. Similarly, when we print the array using `print_intarray`, we give it the array to be printed and a count of elements it contains. We saw in Chapter 6 that in order for a *called* function to access objects in the *calling* function (such as to store elements in an array) we must use *indirect access*, i.e. pointers. So, `read_intaray()` must indirectly access the array, `exam_scores`, in `main()`. One unique feature of C is that array access is *always* indirect; thus making it particularly easy for a called function to indirectly access elements of an array and store or retrieve values. As we will see in later sections, array access by index value is interpreted as an indirect access, so we may simply use array indexing as indirect access.

We are now ready to implement the algorithm for `main()` using functions to read data into the array and to print the array. The code is shown in Figure 7.5. The function calls in `main()` pass the name of the array, `exam_scores`, as an argument because the name of an array in an expression evaluates to a pointer to the array. In other words, the expression, `exam_scores`, is a *pointer* to (the first element of) the array, `exam_scores[]`. Its type is, therefore, `int *`, and a called function uses this pointer (passed as an argument) to indirectly access the elements of the array. As seen in the Figure, for both functions, the headers and the prototypes show the first formal parameter as an integer array without specifying the size. In C, this syntax is interpreted as a pointer variable; so `scores` is declared aa an `int *` variable. We will soon discuss how arrays are accessed in C; for now, we will assume that these pointers may be used to indirectly access the arrays.

The second formal parameter in both functions is `lim` which specifies the maximum number of items. For `read_intaray()`, this may be considered the maximum number of scores that can be read so that it does not read more items than the size of the array allows (`MAX`). The function returns the *actual* number of items read which is saved in the variable, `n`, in `main()`. For the function, `print_intaray()`, `lim` represents the fact that it must not print more than `n` items. Again, since arrays in C are accessed indirectly, these functions are able to access the array which is defined and allocated in `main()`. A sample session for this implementation of the task would be identical to the one shown earlier.

Similarly, we can modify the program, `string.c`, to use functions to read and print strings. The task and the algorithm are the same as defined for STRING0 in the last section, except that the program is terminated when an empty string is read. The code is shown in Figure 7.6. The driver calls `read_str()` and `print_str()` repeatedly until an empty string is read (detected when `s[0]` is zero, i.e. `NULL`). The argument passed to `read_str()` and `print_str()` is `str`, a pointer to (the first element of) a character array, i.e. a `char *`. The function, `read_str()`, reads characters until a newline is read and indirectly stores the characters into the string, `s`. The function, `print_str()`, prints characters from the string, `s` until `NULL` is reached and terminates the output with a newline. Notice we have declared the formal parameter, `s` as a `char *`, rather than as an array: `char s[]`. As we will see in the next section, C treats the two declarations exactly the same.

```
/*  File: scores2.c
    This program uses functions to read scores into an array and to print
    the scores.
*/
#include <stdio.h>
#define MAX 10

int read_intaray(int scores[], int lim);
print_intaray(int scores[], int lim);
main()
{   int n, exam_scores[MAX];

    printf("***List of Exam Scores***\n\n");
    n = read_intaray(exam_scores, MAX);
    print_intaray(exam_scores, n);
}

/* Function reads scores in an array. */
int read_intaray(int scores[], int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");

    while ((count < lim) && (scanf("%d", &n) != EOF)) {
         scores[count] = n;
         count++;
    }
    return count;
}

/* Function prints lim elements in the array scores. */
void print_intaray(int scores[], int lim)
{   int i;

    printf("\n***Exam Scores***\n\n");
    for (i = 0; i < lim; i++)
         printf("%d\n", scores[i]);
}
```

Figure 7.5: Code fore `scores.c`

```
/* File: string2.c
   This program reads and writes strings until an empty string is
   read. It uses functions to read and print strings to standard
   files.
*/
#include <stdio.h>
#define SIZE 100
void print_str(char s[]);
void read_str(char s[]);

main()
{  char str[SIZE];

   do {
     read_str(str);
     print_str(str);
   } while (str[0]);
}

/* Function reads a string from standard input until a newline is
   read. A NULL is appended.
*/
void read_str(char *s)
{  int i;
   char c;

   for (i = 0; (c = getchar()) != '\n'; i++)
     s[i] = c;
   s[i] = NULL;
}

/* Function prints a string to standard output and terminates with a
   newline.
*/
void print_str(char *s)
{  int i;

   for (i = 0; s[i]; i++)
     putchar(s[i]);
   putchar('\n');
}
```

Figure 7.6: Code for `string2.c`

## 7.3 Arrays, Pointers, Pointer Arithmetic

Let us now examine how arrays are actually accessed in C. As we have seen, an array is a sequence of objects, each of the same data type. The starting address of this array of objects, i.e. the address of the first object in the array is called the **base address** of the array. The address of each successive element of the array is offset from the base by the size of the array type, e.g. for each successive element of an integer array, the address is offset by the size of an integer type object. As we mentioned in the previous section, in C, the name of an array used by itself in an expression evaluates to the base address of the array. That is, this value is a *pointer* type and points to the first object of the array. The name of the array is said to point to the array. Figure 7.7 shows an array, `X[]` with `X` pointing to (the first object of) the array. If the array is an integer array, (float array, character array, etc.) then the type of `X` is `int *` (`float *`, `char *`, etc.). Thus, the declaration of an array causes the compiler to allocate the specified number of contiguous cells of the indicated type, as well as to allocate an appropriate pointer cell, initialized to point to the first cell of the array. This pointer cell is given the name of the array. Since `X` points to `X[0]`, the following are equivalent:

```
X <---> &X[0]
```

Thus, the dereferenced pointer, `*X`, accesses the object, `X[0]`, i.e. the following are equivalent:

```
*X <---> X[0]
```

As we have seen, pointer variables point to objects of a specific type. We might suspect that they can be increased or decreased to point to contiguous successive or preceding objects of the same type. In C, adding one to a pointer makes the resulting pointer point to the next object of the same type. (The value of the new pointer equals the original value of the pointer increased by the size of the object pointed to). For the array above, `X + 1` points to `X[1]`; the increase in the pointer value is made by the appropriate size of the type involved. For example, if `X` is an integer array and an integer requires 4 bytes, then the value of `X + 1` will be greater than that of `X` by 4. Adding $k$ to a pointer results in a pointer to a successive object offset by $k$ objects from the original. Thus, `X + 0` points to the start of the array (the first element, `X[0]`), `X + 1` points to the next element, `X[1]`, and `X + k` points to `X[k]` as can be seen in Figure 7.7. Similarly, `&X[k]` is the same as `X + k`, and `X[k]` is the same as `*(X + k)`. Table 7.1 summarizes pointer arithmetic and indirect access of elements of an array. Pointer arithmetic may also involve subtraction; the resulting pointer points to a previous object offset appropriately. Thus, for example, `&X[3] - 1` points to `X[2]`, `&X[5] - 3` points to `X[2]`, and so on.

In C array access is always made through pointers and indirection operators. Whenever an expression such as `X[k]` appears in a program, the compiler interprets it to mean `*(X + k)`. In other words, objects of an array are always accessed indirectly. As we have seen previously, this makes it particularly easy for a called function to indirectly access elements of an array allocated in the calling function to store or retrieve values. Let us see how function calls handle array access using the program, `scores2.c` of the last section. The relevant function calls in `main()` and the corresponding function headers are shown below for easy reference:

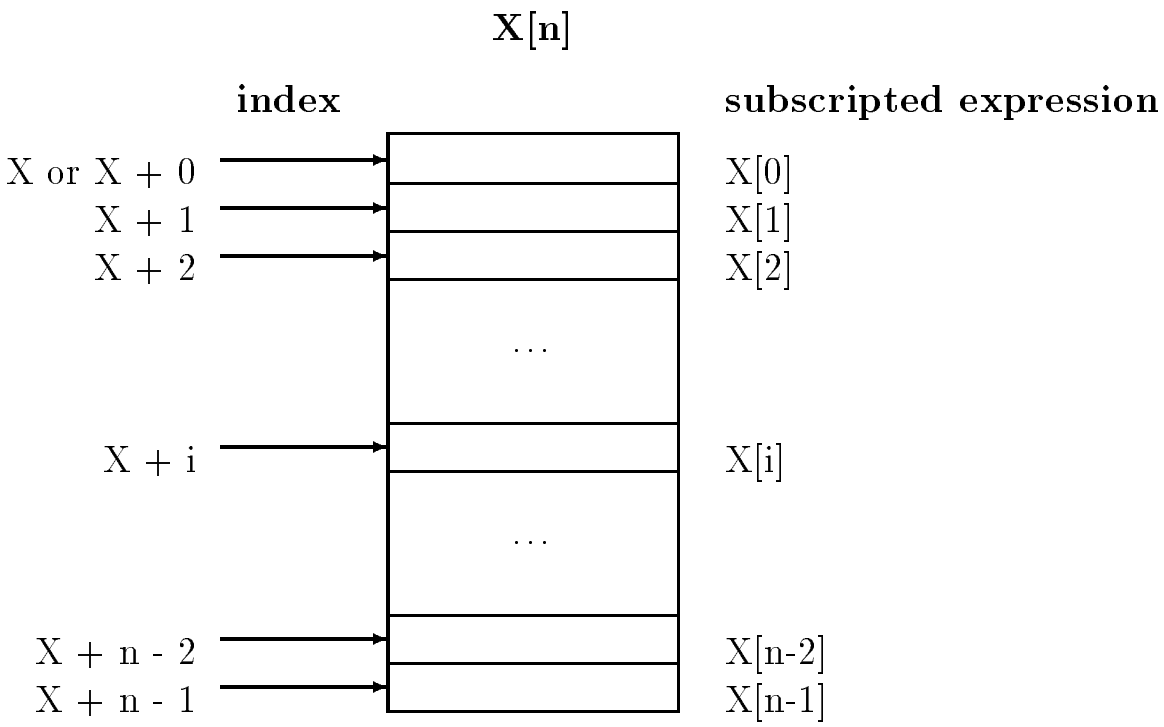


Figure 7.7: Pointer Arithmetic

| Pointer Arithmetic | Address of Operator | Array Subscripting | Indirect Reference |
|---|---|---|---|
| X + 0 | &X[0] | X[0] | *(X + 0) |
| X + 1 | &X[1] | X[1] | *(X + 1) |
| X + 2 | &X[2] | X[2] | *(X + 2) |
| X + 3 | &X[3] | X[3] | *(X + 3) |
| ... | ... | ... | ... |
| X + k | &X[k] | X[k] | *(X + k) |

Table 7.1: Pointer Arithmetic and Indirect Access

```
main()
{   int exam_scores[MAX];
    ...
    n = read_intaray(exam_scores, MAX);
    print_intaray(exam_scores, n);
}
int read_intaray(int scores[], int lim)
{
    ...
}
void print_intaray(int scores[], int lim)
{
    ...
}
```

When a formal parameter is declared in a function header as an array, it is interpreted as a pointer variable, NOT an array. Even if a size were specified in the formal parameter declaration, only a pointer cell is allocated for the variable, not the entire array. The type of the pointer variable is the specified type. In our example, the formal parameter, `scores`, is an integer pointer. It is initialized to the pointer value passed as an argument in the function call. The value passed in `main()` is `exam_scores`, a pointer to the first element of the array, `exam_scores[]`. Figure 7.8 illustrates the connection between the calling function, `main()`, and the called function, `read_intaray()`. In this case, the formal parameter, `scores`, is initialized to point to the value of `exam_scores` which is a pointer to (the first element of) the array `exam_scores[]`. The Figure also shows that `lim` is initialized to 10.

Within the function, `read_scores()`, it is now possible to access all the elements of the array, `exam_scores[]`, indirectly. Since the variable, `scores`, in `read_intaray()` points to the first element of the array, `exam_scores[]`, `*scores` accesses the first element of the array, i.e. `exam_scores[0]`. In addition, `scores + 1` points to the next element of the array, so `*(scores + 1)` accesses the next element, i.e. `exam_scores[1]`. In general, `*(scores + count)` accesses the element `exam_scores[count]`. To access elements of the array, we can either write `*(scores + count)` or we can write `scores[count]`, because dereferenced array pointers and indexed array elements are identical ways of writing expressions for array access.

The functions, `read_intaray()` and `print_intaray()` can be used to read objects into *any* integer array and to print element values of *any* integer array, respectively. The calling function must simply pass, as arguments, an appropriate array pointer and maximum number of elements.

These functions may also be written explicitly in terms of indirect access, for example:

```
/* Function reads scores in an array. */
int read_intaray2(int * scores, int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");
```

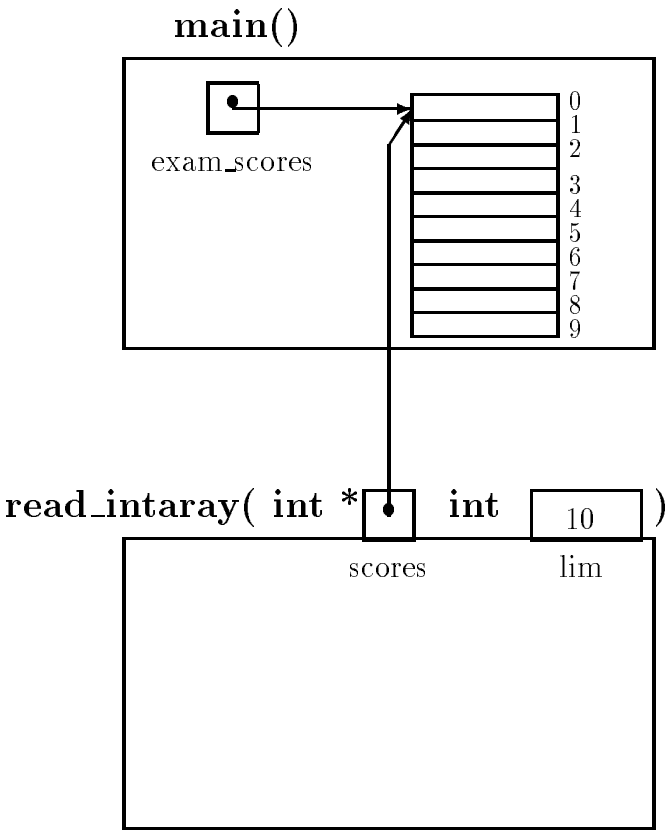


Figure 7.8: Array Pointers as Function Parameters

```
    while ((count < lim) && (scanf("%d", &n) != EOF)) {
         *(scores + count) = n;
         count++;
    }
    return count;
}
```

Alternatively, since `scores` is a pointer variable, we can increment its value each time so that it points to the next object of integer type in the array, such as:

```
/* Function reads scores in an array. */
int read_intaray3(int * scores, int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");
    while ((count < lim) && (scanf("%d", &n) != EOF)) {
         *scores = n;
         count++;
         scores++;
    }
    return count;
}
```

The first time the loop is executed, `*scores` accesses the element of the array at index 0. The *local* pointer cell, `scores`, is then incremented to point to the next element of the array, at index 1. The second time the loop is executed, *scores accesses the array element at index 1. The process continues until the loop terminates.

It is also possible to mix dereferenced pointers and array indexing:

```
/* Function reads scores in an array. */
int read_intaray4(int scores[], int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");
    while ((count < lim) && (scanf("%d", &n) != EOF)) {
         *(scores + count) = n;
         count++;
    }
    return count;
}
```

or:

```
/* Function reads scores in an array. */
```

```
int read_intaray5(int * scores, int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");
    while ((count < lim) && (scanf("%d", &n) != EOF)) {
         scores[count] = n;
         count++;
    }
    return count;
}
```

We can also consider *parts* of an array, called a **sub-array**. A pointer to a sub-array is also an array pointer; it simply specifies the *base* of the sub-array. In fact, as far as C is concerned, there is no difference between an entire array and any of its sub-arrays. For example, a function call can be made to print a sub-array by specifying the starting pointer of the sub-array and its size. Suppose we wish to print the sub-array starting at `exam_scores[3]` containing five elements; the expression, `&exam_scores[3]` is a pointer to an array starting at `exam_scores[3]`. The function call is:

```
print_intaray(&exam_scores[3], 5);
```

Alternately, since `exam_scores + 3` points to `exam_scores[3]`, the function call can be:

```
print_intaray(exam_scores + 3, 5);
```

The passed parameters are shown visually in Figure 7.9. If either of the above function calls is used in the program, `scores2.c`, the values of `exam_scores[3]`, `exam_scores[4]`, ..., and `exam_scores[7]` will be printed.

### 7.3.1 Pointers: Increment and Decrement

We have just seen that an array name, e.g. `aa`, is a pointer to the array and that `aa + i` points to `aa[i]`. We can illustrate this point in the program below, where the values of pointers themselves are printed. A pointer value is a byte address and is printed as an unsigned integer (using conversion specification for unsigned integer, `%u`). The program shows the relationships between array elements, pointers, and pointer arithmetic.

```
/* File: arayptr.c
   This program shows the relation between arrays and pointers.
*/
#include <stdio.h>
#define N 5
```

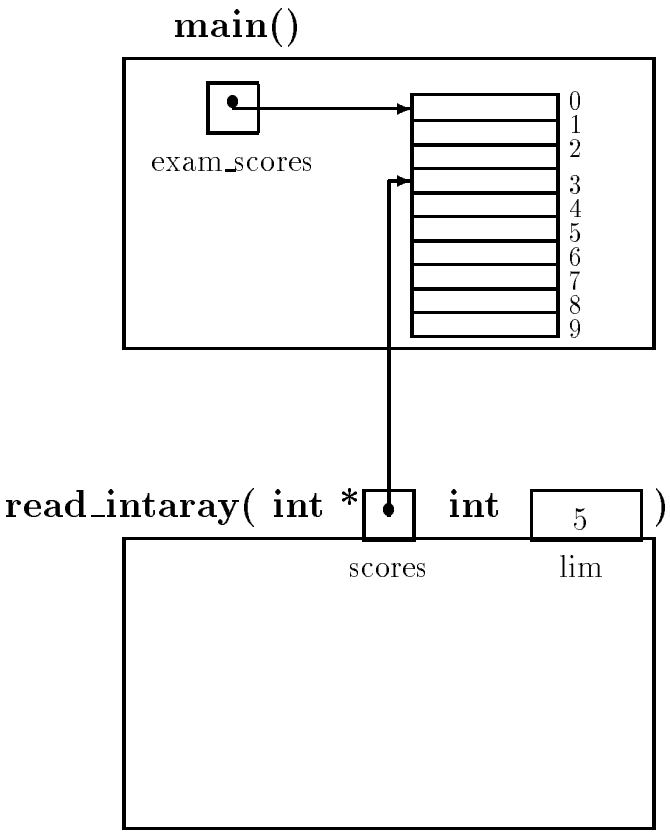


Figure 7.9: Pointer to a Sub-array

```
main()
{   int i, j, aa[N];

    printf("***Pointers, Arrays, and Pointer Arithmetic***\n\n");

    for (i = 0; i < N; i++) {
         aa[i] = i * i;
         printf("aa + %d = %u; &aa[%d] = %u\n", i, aa + i, i, &aa[i]);
         printf("*(aa + %d) = %d; aa[%d] = %d\n", i, *(aa + i), i, aa[i]);
    }
}
```

In the loop, we first assign a value to each `aa[i]`. We then print values to show that pointers, `aa + i` and `&aa[i]` are the same, i.e. that `aa + i` points to `aa[i]`. Next, we print the array element values to show that `*(aa + i)` is the same as `aa[i]`. A sample output for the program is shown below:

```
***Pointers, Arrays, and Pointer Arithmetic***

aa + 0 = 65480; &aa[0] = 65480
*(aa + 0) = 0; aa[0] = 0
aa + 1 = 65482; &aa[1] = 65482
*(aa + 1) = 1; aa[1] = 1
aa + 2 = 65484; &aa[2] = 65484
*(aa + 2) = 4; aa[2] = 4
aa + 3 = 65486; &aa[3] = 65486
*(aa + 3) = 9; aa[3] = 9
aa + 4 = 65488; &aa[4] = 65488
*(aa + 4) = 16; aa[4] = 16
```

(In the host implementation where the above program was executed, two bytes are required for integers; therefore, successive array element addresses are two bytes apart).

The next example shows that pointers may be incremented and decremented. In either case, if the original pointer points to an object of a specific type, the new pointer points to the next or previous object of the same type, i.e. pointers are incremented or decremented in steps of the object size that the pointer points to. Thus, it is possible to traverse an array starting from a pointer to any element in the array. Consider the code:

```
/*  File: arayptr2.c
    Pointers and pointer arithmetic.
*/
#include <stdio.h>
#define N 5
```

```
main()
{   float faray[N], *fptr;
    int *iptr, iaray[N], i;

    /* initialize */
    for (i = 0; i < N; i++) {
         faray[i] = 0.3;
         iaray[i] = 1;
    }

    /* initialize fptr to point to element faray[3] */
    fptr = &faray[3];
    *fptr = 1.;              /* faray[3] = 1. */
    *(fptr - 1) = .9;        /* faray[2] = .9 */
    *(fptr + 1) = 1.1;       /* faray[4] = 1.1 */

    /* initialize iptr in the same way */
    iptr = &iaray[3];
    *iptr = 0;
    *(iptr - 1) = -1;
    *(iptr + 1) = 2;

    for (i = 0; i < N; i++) {
         printf("faray[%d] = %f   ", i, *(faray + 1));
         printf("iaray[%d] = %d\n", i, iaray[i]);
    }
}
```

The program is straightforward. It declares a float array of size 5, and an integer array of the same size. The float array elements are all initialized to 0.3, and the integer array elements to 1. The program also declares two pointer variables, one a float pointer and the other an integer pointer. Each pointer variable is initialized to point to the array element with index 3; for example, `fptr` is initialized to point to the float array element, `faray[3]`. Therefore, `fptr - 1` points to `faray[2]`, and `fptr + 1` points to `faray[4]`. The value of `*fptr` is then modified, as is the value of `*(fptr - 1)` and `*(fptr + 1)`. Similar changes are made in the integer array. Finally, the arrays are printed. Here is the output of the program:

```
faray[0] = 0.300000 iaray[0] = 1
faray[1] = 0.300000 iaray[1] = 1
faray[2] = 0.900000 iaray[2] = -1
faray[3] = 1.000000 iaray[3] = 0
faray[4] = 1.100000 iaray[4] = 2
```

### 7.3.2 Array Names vs Pointer Variables

As we have seen, when we declare an array, a contiguous block of memory is allocated for the cells of the array and a pointer cell (of the appropriate type) is also allocated and initialized to point to the first cell of the array. This pointer cell is given the name of the array. When memory is allocated for the array cells, the starting address is fixed, i.e. it cannot be changed during program execution. Therefore, the value of the pointer cell should not be changed. To ensure that this pointer is not changed, in C, array names may not be used as variables on the left of an assignment statement, i.e. they may not be used as an Lvalue. Instead, if necessary, separate pointer variables of the appropriate type may be declared and used as Lvalues. For example, we can use pointer arithmetic and the dereference operator to initialize an array as follows:

```
/* Use of pointers to initialize an array */
#include <stdio.h>
main()
{   int i;
    float X[MAX];

    for (i = 0; i < MAX; i++)
         *(X + i) = 0.0;     /* same as X[i] */
}
```

In the loop, `*(X + i)` is the same as `X[i]`. Since `X` (the pointer cell) has a fixed value we cannot use the increment operator or the assignment operator to change the value of `X`:

```
X = X + 1;     /* ERROR */
```

Here is an example of a common error which attempts to use an array as an Lvalue:

```
/* BUG: Attempt to use an array name as an Lvalue */
#include <stdio.h>
main()
{   int i;
    float X[MAX];

    for (i = 0; i < MAX; i++)  {
         *X = 0.0;
         X++;      /* BUG: X = X + 1; */
    }
}
```

In this example, `X` is fixed and cannot be used as an Lvalue; the compiler will generate an error stating that an Lvalue is required for the `++` operator. However, we can declare a pointer variable,

which can point to the same type as the type of the array, and initialize it to the value of array pointer. This pointer variable CAN be used as an Lvalue, so we can then rewrite the previous array initialization loop as follows:

```
/*  OK: A pointer variable is initialized to an array pointer and then
    used as an Lvalue.
*/
#include <stdio.h>
main()
{   int i;
    float *ptr, X[MAX];

    ptr = X;       /* ptr is a variable which can be assigned a value */
    for (i = 0; i < MAX; i++) {
         *ptr = 0.0;         /* *ptr accesses X[i] */
         ptr++;
    }
}
```

Observe that the pointer variable, `ptr`, is type `float *`, because the array is of type `float`. It is initialized to the value of the fixed pointer, `X` (i.e. the initial value of `ptr` is set to the same as that of `X`, namely, `&X[0]`), and may subsequently be modified in the loop to traverse the array. The first time through the loop, `*ptr` ( `X[0]`) is set to zero and `ptr` is incremented by one so that it points to the next element in the array. The process repeats and each element of the array is set to 0.0. This behavior is shown in Figure 7.10. Observe that the final increment of `ptr` makes it point to `X[MAX]`; however, no such element exists (recall, an array of size `MAX` has cells indexed 0 to `MAX - 1`). At the end of the `for` loop, the value of `ptr` is meaningless since it now points outside the array. Unfortunately, C does not prevent a program from accessing objects outside an array boundary; it merely increments the pointer value and accesses memory at the new address. The results of accessing the array with the pointer, `ptr` at this point will be meaningless and possibly disastrous. It is the responsibility of the programmer to make sure that the array boundaries are not breached. The best way of ensuring that a program stays within array boundaries is to write all loops that terminate when array limits are reached. When passing arrays in function calls, always pass the array limit as an argument as well.

Here is a similar error in handling strings and pointers:

```
/* BUG: Common error in accessing strings */
#include <stdio.h>
#define SIZE 100
main()
{   char c, msg[SIZE];

    while ((c = getchar()) != '\n') {
         *msg = c;
```

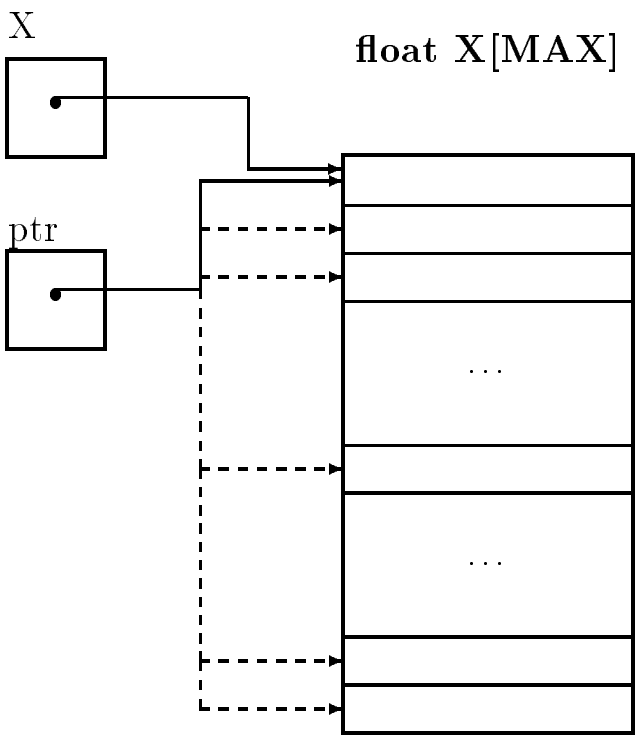


Figure 7.10: Pointer Variables and Arrays

```
        msg++;          /* msg is fixed; it cannot be an Lvalue */
    }
    *msg = '\0';
}
```

The array name, `msg` is a constant pointer; it cannot be used as an `Lvalue`. We can rewrite the loop correctly to read a character string as:

```
/* OK: Correct use of pointers to access a string */
#include <stdio.h>
#define SIZE 100
main()
{   char c, *mp, msg[SIZE];

    mp = msg;
    while ((c = getchar()) != '\n') {
         *mp = c;
         mp++;          /* mp is a variable; it can be an Lvalue */
    }
    *mp = '\0';
}
```

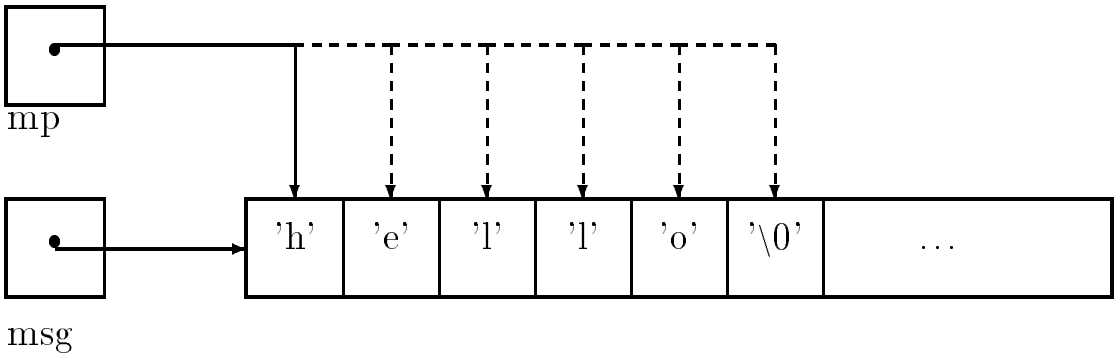


Figure 7.11: Pointer Variables and Strings

Observe in this case, `mp` is a character pointer since the array is a character array. The variable, `mp` is initialized to the value of `msg`. The dereferenced pointer variable, `*mp`, then accesses the elements of the array in sequence as `mp` is incremented (see Figure 7.11). The loop terminates when a newline is read, and a terminating `NULL` is added to the string.

Remember, pointer variables must be initialized to point to valid objects; otherwise, fatal errors will most likely occur. For example, if the pointer, `mp`, in the above code were not initialized to the value of `msg`, a serious and probably fatal error will occur when the pointer is dereferenced and an attempt is made to access the memory cell pointed to by `mp`. This is because the initial value of `mp` would be some garbage value which may point to an invalid memory address causing a fatal memory fault to occur. If the garbage value were not an invalid memory address, the loop would write characters to an unknown memory address, possibly destroying other valid data.

As we've said, an array names cannot be used as an Lvalues. On the other hand, when a function is used to access an array, the corresponding formal parameter is a pointer variable. This pointer variable can be used as an Lvalue. Here is a function to print a string:

```
/* Function prints a string pointed to by mp. */
void our_strprint(char *mp)
{
    while (*mp) {
         putchar(*mp);
         mp++;          /* mp is a variable; it can be an Lvalue */
    }
    putchar('\n');
}
```

Here, `mp` is a pointer variable, which, when the function is called, we assume will be initialized to point to some `NULL` terminated string. The expression, `*mp`, accesses the elements of the array, and the loop continues as long as `*mp` is not `NULL`. Each time the loop is executed, a character, `*mp`, is written, and `mp` is incremented to point to the next character in the array. When `*mp` accesses the `NULL`, the loop terminates and a newline character is written.

## 7.4 String Assignment and I/O

As we have seen, a character string in C is an array of characters with a terminating `NULL` character. Access to a character string requires only a pointer to the character array containing the characters. It is common to use the term, **string**, to loosely refer to either an array of characters holding the string, or to a character pointer that may be used to access the string; it should be clear from context which is meant.

When a character *string constant* is used in a program, the compiler automatically allocates an array of characters, stores the string in the array, appends the `NULL` character, and replaces the string constant by the value of a pointer to the string. Therefore, the value of a string constant is the value of a pointer to the string. We can use string constants in expressions just as we can use the names of arrays. Here is an example:

```
char *mp, msg[SIZE];

mp = "This is a message\n";
```

The compiler replaces the string constant by a pointer to a corresponding string. Since `mp` is a character pointer variable, we can assign a value of a fixed string pointer to `mp`. If necessary we can traverse and print the string using this pointer. On the other hand, since `msg[]` is declared as a character array, we cannot make the following assignment:

```
msg = "This is a message\n";       /* ERROR */
```

since we are attempting to modify a *constant* pointer, `msg`.

A string constant is just another string appropriately initialized and accessed by a pointer to it. We will therefore make no distinctions between strings and string constants; they are both strings referenced by string pointers. While strings and string constants are both strings, the contents of string constants cannot be changed in ANSI C.

We have been using string constants as format strings for `printf()` and in `scanf()`, which expect their first argument to be a string pointer; i.e. a `char` pointer. The compiler has automatically created an appropriate string and replaced the string by a string pointer. Instead of writing a format string directly in a function call, we could pass a string pointer that points to a format string. Here is an example:

```
char *mesg;
int n;

n = 1;
mesg = "This is message number %d\n";
printf(mesg, n);
```

The string constant is stored by the compiler somewhere in memory as an array of characters with an appended `NULL` character. A pointer to this character array is assigned to the character pointer variable, `mesg`. The function `printf()` then uses the pointer to retrieve the format string, and print the string:

```
This is message number 1
```

The functions, `printf()` and `scanf()` can be used for string input and output as well. Array names or properly initialized pointers to strings must be passed as arguments in both cases. The conversion specification for strings is `%s`. For example, consider the task of reading strings and writing them out. Here is an example program.

```
/*  File: fcopy.c
    This program reads strings from standard input using scanf() and writes
    them to standard output using printf().
*/
#include <stdio.h>
#include "araydef.h"
main()
{   char mesg[SIZE];

    printf("***Strings: Formatted I/O***\n\n");
    printf("Type characters, EOF to quit\n");
    while (scanf("%s", mesg) != EOF)
         printf("%s\n", mesg);
}
```

Sample Session:

```
***Strings:  Formatted I/O***

Type characters, EOF to quit
        This      is       a       test
This
is
a
test
^D
```

The conversion specification, `%s` indicates a string and the corresponding matching argument must be a char pointer. When `scanf()` reads a string it stores it at the location pointed to by `mesg` — note we do not use `&mesg` since `mesg` is already a pointer to an array of characters. Then, `printf()` prints the string pointed to by `mesg`. When `scanf()` reads a string using `%s`, it behaves like it

does for numeric input, skipping over leading white space, and reading the string of characters until a white space is reached. Thus, `scanf()` can read only single words, storing the string of characters read into an array pointed to by the argument, `mesg` and appending a `NULL` character to mark the end of the string. On the other hand, `printf()` prints the string pointed to by its argument, `mesg`, printing the entire string (including any white space) until a `NULL` character is reached. The sample session shows that each time `scanf()` reads a string, only a single word is read from the input line and then printed.

As we said, when `scanf()` reads a string, the string argument must be a pointer that points to an array where the input characters are to be stored. For example, here are correct and incorrect ways of using `scanf()`:

```
char * mp, * mptr, msg[SIZE];

scanf("%s", mp);    /* BUG */
scanf("%s", msg);   /* OK */
mptr = msg;
scanf("%s", mptr);  /* OK */
```

The first `scanf()` is incorrect because `mp` has not been initialized and, therefore, does not point to an array where a string is to be stored. The other statements are correct; in each case, the pointer points to an array.

## 7.5 Array Initializers

ANSI C allows automatic array variables to be initialized in declarations by constant initializers as we have seen we can do for scalar variables. These initializing expressions must be *constant* (known at compile time) values; expressions with identifiers or function calls may not be used in the initializers. The initializers are specified within braces and separated by commas. Here are some declarations with constant initializers:

```
int ex[10] = { 12, 23, 9, 17, 16, 49 };
char word[10] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

Each constant initializer in the braces is assigned, in sequence, to an element of the array starting with index 0. If there are not enough initializers for the whole array, the remaining elements of the array are initialized to zero. Thus, `ex[0]` through `ex[5]` are assigned the values 12, 23, 9, 17, 16, and 49, while `ex[6]` through `ex[9]` are initialized to zero. Similarly, `word` is initialized to a string `"hello"`. String initializers may be written as string constants instead of character constants within braces, for example:

```
char mesg[] = "This is a message";
char name[20] = "John Doe";
```

In the case of `mesg[]`, enough memory is allocated to accommodate the string plus a terminating `NULL`, and we do not need to specify the size of the array. The above string initializers are allowed as a convenience, the compiler initializes the array at compile time. Remember, initializations are *not* assignment statements; they are declarations that allocate and initialize memory. As with other arrays, these array names cannot be used as Lvalues in assignment statements.

Here is a short program that shows the use of initializers:

```
/* File: init.c
   Program shows use of initializers for arrays.
*/
#include <stdio.h>
#define MAX 10
main()
{   int i, ex[MAX] = { 12, 23, 9, 17, 16, 49 };
    char word[MAX] = {'S', 'm', 'i', 'l', 'e', '\0'};
    char mesg[] = "Message of the day is: ";

    printf("***Array Declarations with Initializers***\n\n");
    printf("%s%s\n", mesg, word);
    printf("Initialized Array:\n");
    for (i= 0; i < MAX; i++)
         printf("%d\n", ex[i]);
}
```

Sample Output:

```
***Array Declarations with Constant Initializers***

Message of the day is:  Smile
12
23
9
17
16
49
0
0
0
0
```

The first `printf()` statement uses `%s` to print each of the two strings accessed by pointers, `mesg` and `word`. Finally, the loop prints the array, `ex`, one element per line.

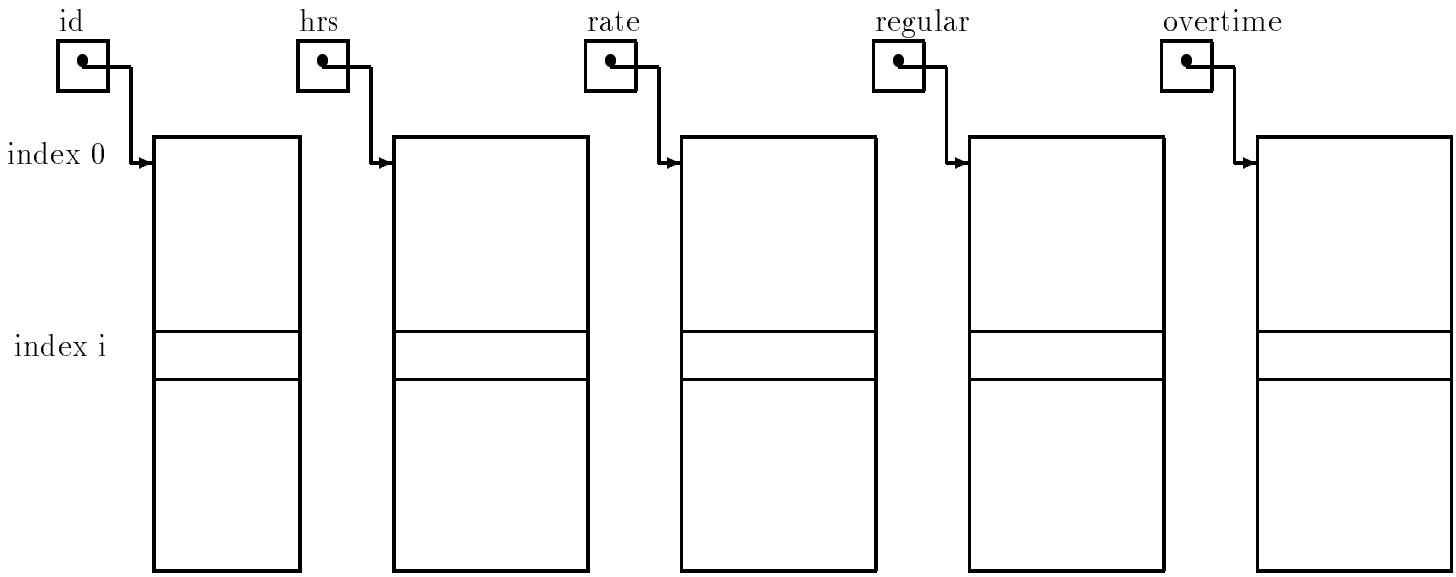


Figure 7.12: Data Record Spread Over Several Arrays

## 7.6 Arrays for Databases

We now consider our payroll task that reads input data and calculates pay as before, but the program prints a table of computed pay for all the id's. The algorithm uses arrays to store the data, but is otherwise very similar to our earlier programs: get data, calculate pay, and print results. We will use functions to perform these subtasks. Here are the prototypes:

```
/* File: payutil.h */
int getdata(int id[], float hrs[], float rate[], int lim);
void calcpay(float hrs[], float rate[], float reg[], float over[], int n);
void printdata(int id[], float hrs[], float rate[],
                              float reg[], float over[], int n);
```

The function, `getdata()` gets the data into the appropriate arrays for id's, hours worked, and rate of pay; returning the number of id's entered by the user. While the arrays `id[]`, `hrs[]`, and `rate[]` are individual arrays, we make sure that the same value of the array index accesses the data for a given id. For example, `id[i]` accesses an id number and `hrs[i]` and `rate[i]` access hours worked and rate of pay for that id number. In other words, an input *data record* for each id number resides at the same index in these arrays.oWe can think of this *data structure* as a table, where the columns are the arrays holding different pieces of information; and the rows are the data for an individual id, as shown in Figure 7.12.

Next, `calcpay()` calculates and stores regular and overtime pay for each id in arrays, `regpay[]` and `overpay[]` (columns), at the same array index as the input data record. Thus, the entire payroll data record for each id number is at a unique index in each of the arrays. Finally, `printdata()`

```
/*  File: paytab.c
    Other Source Files: payutil.c
    Header FIles: paydef.h, payutil.h
    Program calculates and stores payroll data for a number of id's. Gets
    data, calculates pay, and prints data for all id's.
*/
#include <stdio.h>
#include "paydef.h"
#include "payutil.h"
#define MAX 10

main()
{   int n, id[MAX];
    float hrs[MAX], rate[MAX], regpay[MAX], overpay[MAX];

    printf("***Payroll Program***\n\n");
    n = getdata(id, hrs, rate, MAX);
    calcpay(hrs, rate, regpay, overpay, n);
    printdata(id, hrs, rate, regpay, overpay, n);
}
```

Figure 7.13: Code for `paytab.c`

prints each payroll record, i.e. the input data as well as the calculated regular, overtime, and total pay. We will write `getdata()`, `printdata()`, and `calcpay()` in the file, `payutil.c`. The prototypes shown above for these functions are in the file, `payutil.h`. This header file is included in the program file `paytab.c`, where `main()` will reside (see Figure 7.13). We also include the header file, `paydef.h` which defines the symbolic constants, `REG_LIMIT` and `OT_FACTOR`.:

```
/* paydef.h */
#define REG_LIMIT 40
#define OT_FACTOR 1.5
```

The program calls `getdata()` which reads data into the appropriate arrays and stores the returned value (the number of id's) into `n`. It then calls on `calcpay()` to calculate the pay for `n` people, filling in the `regpay[]` and `overpay[]` arrays, and calls `printdata()` to print the input data and the results for `n` people. The code for these functions is shown in Figure 7.14.

In the function, `getdata()`, `scanf()` is used to read data for the itemsa, using `n` to count and index the data items in the arrays. We use pointer arithmetic to pass the necessary arguments to `scanf()`. For example, to read data into `id[n]`, we must pass its address `&id[n]`. Instead, we pass `id + n` which is identical to `&id[n]`. The function, `getdata()`, reads data for as many id's as possible, returning either when there is no more data (a zero id value) or the arrays are full (`n` reaches the limit, `lim` passed in). If the array limit is reached, an appropriate message is printed

```
/* File: payutil.c */
#include <stdio.h>
#include "paydef.h"
#include "payutil.h"
/* Gets data for all valid id's and returns the number of id's */
int getdata(int id[], float hrs[], float rate[], int lim)
{   int n = 0;
    while (n < lim) {
         printf("ID <zero to quit>: ");
         scanf("%d", id + n);          /* id + n  is same as &id[n] */
         if (id[n] <= 0) return n;
         printf("Hours Worked: ");
         scanf("%f", hrs + n);         /* hrs + n is same as &hrs[n] */
         printf("Rate of Pay: ");
         scanf("%f", rate + n);        /* rate + n is same as &rate[n] */
         n++;
    }
    printf("No more space for data - processing data\n");
    return n;
}
/* Calculates regular and overtime pay for each id */
void calcpay(float hrs[], float rate[], float reg[], float over[], int n)
{   int i;
    for (i = 0; i < n; i++) {
         if (hrs[i] <= REG_LIMIT) {
              reg[i] = hrs[i] * rate[i];
              over[i] = 0;
         }
         else {
              reg[i] = REG_LIMIT * rate[i];
              over[i] = (hrs[i] - REG_LIMIT) * OT_FACTOR * rate[i];
         }
    }
}

/* Prints a table of payroll data for all id's. */
void printdata(int id[], float hrs[], float rate[],
                        float reg[], float over[], int n)
{   int i;
    printf("***PAYROLL: FINAL REPORT***\n\n");
    printf("%4s\t%5s\t%5s\t%6s\t%6s\t%6s\n", "ID", "HRS", "RATE",
              "REG", "OVER", "TOT");
    for (i = 0; i < n; i++)
         printf("%4d\t%5.2f\t%5.2f\t%6.2f\t%6.2f\t%6.2f\n",
                   id[i], hrs[i], rate[i], reg[i], over[i],
                   reg[i] + over[i]);
}
```

Figure 7.14: Code for `payutil.c`

and the input is terminated. The function returns the number of id's placed in the arrays. The other functions in the program are straight forward; each index accesses the data record for the id at that index.

As written, `getdata()` terminates input of data when an invalid id (`id <= 0`) is entered. An alternative might be to read a data item in a temporary variable first, examine it for validity if desired, and then assign it to an array element. For example:

```
scanf("%d", &x);
if (x > 0)
    id[n] = x;
else
    return n;
```

Here is a sample session for the program, `paytab.c` compiled and linked with `payutil.c`:

Sample Session:

```
***Payroll Program***

ID <zero to quit>:  8
Hours Worked:  50
Rate of Pay:  14
ID <zero to quit>:  12
Hours Worked:  45
Rate of Pay:  12.50
ID <zero to quit>:  2
Hours Worked:  20
Rate of Pay:  5
ID <zero to quit>:  5
Hours Worked:  40
Rate of Pay:  10
ID <zero to quit>:  0
***PAYROLL: FINAL REPORT***


 ID    HRS   RATE     REG    OVER     TOT

  8  50.00  14.00  560.00  210.00  770.00
 12  45.00  12.50  500.00   93.75  593.75
  2  20.00   5.00  100.00    0.00  100.00
  5  40.00  10.00  400.00    0.00  400.00
```

## 7.7 Common Errors

1. Use of an array name as an Lvalue: An array name has a fixed value of the address where the array is allocated. It is NOT a variable; it cannot be used as an Lvalue and assigned a new value. Here are some example:

   (a)
   ```
   int x[10];

   while (*x) {
       ...
       x++;      /* ERROR */
   }
   ```
   x cannot be used as an Lvalue and assigned new values.

   (b)
   ```
   char msg[80];
   ...
   while (*msg) {
       ...
       msg++;    /* ERROR */
   }
   ```
   msg cannot be used as an Lvalue.

   (c)
   ```
   char msg[80];

   msg = "This is a message";   /* ERROR */
   ```
   msg cannot be an Lvalue. The right hand side is not a problem. Value of a string constant is a pointer to an array automatically allocated by the compiler.

   (d)
   ```
   char msg[80] = "This is a message";

   /* OK: array initialized to the string when memory is allocated */
   ```
   A string constant initializer is correct. When memory is allocated for the array, it is initialized to the string constant specified.

2. Failure to define an array: Definition of an array is required to allocate memory for storage of an array of objects. A pointer type allocates memory for a pointer variable, NOT for an array of objects. Suppose, `read_str()` reads a string and stores it where its argument points:

   ```
   int *pmsg;    /* memory allocated for a pointer variable */

   read_str(pmsg);    /* ERROR: memory not allocated for a string */
   ```

   No memory is allocated for a string, i.e. an array of characters. The variable, `pmsg` points to some garbage address; `read_str()` will attempt to store the string at that garbage address. The address may be invalid, in which case there will be a memory fault; a fatal error. Allocate memory for a string with an array declaration:

```
int str[MAX];
read_str(str);
```

3. Array pointer not passed to a called function: If a called function is to store values in an array for later use by the calling function, it should be passed a pointer to an array defined in the calling function. Here is a program with an error.

```
#include <stdio.h>
main()
{   char * p, s[80],
         * get_word(char * s);

    p = get_word(s); /* ERROR: returned pointer points to freed memory */
    puts(p);            /* Prints garbage */
}

char * get_word(char *str)
{   char wd[30];             /* memory allocated for array wd[] */
    int i = 0;
    while (*str == ' ')      /* skip leading blanks */
         ;
    while (*str != ' ')      /* while not a delimiter */
         wd[i++] = *str++;   /* copy char into array wd[] */
    wd[i] = '\0';       /* append a NULL to string in wd[] */
    return wd;               /* return pointer to wd[] */
}                            /* memory for array wd[] is freed */
```

The function, `get_word()` copies a word from a string, `s`, into an automatic array, `wd[]` for which memory is allocated in `get_word()`. When `get_word()` returns, a pointer, `wd`, to the calling function, the memory allocated for `wd[]` is freed for other uses, since it was allocated only for `get_word()`. The data stored in `wd[]` may be overwritten with other data. In the calling function, `p` is assigned an address value which points to freed memory. The function, `puts()`, will print a garbage sequence of characters pointed to by `p`. At times, the memory may not be reused right away and it will print the correct string. At other times, it will print out garbage.

4. Errors in passing array arguments: Only array names, i.e., pointers to arrays, should be passed as arguments. The following are all in ERROR:

```
func(s[]);
func(s[80]);
func(*s);
```

Pointers to arrays, i.e. array names by themselves, should be passed as arguments in function calls. Arguments in the above function calls are not pointers. The first one is meaningless in an expression; the second attempts to pass an element at index 80; the third passes a dereferenced pointer, not the pointer to the array.

5. Errors in declaring formal parameters: Formal parameters referencing arrays in function definitions should be specified to be pointers, not objects of a base type. Consider a function, `init()`, that initializes elements of an integer array to some values. The following is an error:

```
init(int aray)
{
     ...
}
```

The parameter declared is an integer not a pointer to an integer. It should be either of the following:

```
init(int * aray)
```

OR

```
init(int aray[])
```

In either of the above cases, memory for an integer *pointer* is allocated, NOT for a new array of integers.

6. Misinterpreting formal parameter declarations: Even if an array size is specified in a formal parameter, memory is not allocated for an array but for a pointer.

```
init(int aray[10])
```

The above declares `aray` as an integer pointer.

7. Pointers are not initialized:

```
int x, * px;
x = 10;
printf("*px = %d\n", *px);
```

Since value of `px` is garbage, there will be a fatal memory fault when an attempt is made to evaluate `*px`.

## 7.8 Summary

In this Chapter, we have introduced one form of compound data type: the **array**. An array is a block of a number of data items all of the same type allocated in contiguous memory cells. We have seen that, in C, an array may be declared using the syntax:

<type-specifier><identifier>[<size>];

specifying the type of the data items, and the number of elements to be allocated in the array. As we saw, such a declaration in a function causes <size> data items of type <type-specifier> to be allocated in contiguous memory, AND a pointer cell to be allocated of type <type-specifier>* (pointer to <type-specifier>), given the name, <identifier>, and initialized to point to the first cell of the array. More specifically, for a declaration like:

```
int data[100];
```

allocates 100 `int` cells, and an `int *` cell named `data` which is initialized to point to the block of integers.

We saw that the data items in an array can be accessed using an **index**; i.e. the number of the item in the block. Numbering of data items begins with index 0, to the size - 1. We use the index of an element in a **subscripting** expression with syntax:

<identifier>[<expression>]

where <identifier> is the name of the array, and the <expression> in the square brackets ([]) is evaluated to the index value. So, for our previous example, the statement:

```
data[0] = 5;
```

sets the integer value, 5, into the first cell of the array, `data`. While:

```
data[i] = data[i-1];
```

would copy the value from the element with index `i - 1` to its immediate successor in the array.

The data types of the elements of an array may be any scalar data type; `int`, `float`, or `char`. (We will see other types for array elements in later chapters). We have cautioned that, in C, no checking is done on the subscripting expressions to ensure that the index is withing the block of data allocated (i.e. that the subscript is *in bounds*). It is the programmers responsibility to ensure the subscript is in bounds. We have seen two ways of doing this: to keep the value of the limit or the extent of data in the array in a separate integer variable and perform the necessary comparisons, or to mark the last item in the array with a special value. The most common use of this later method is in the case of an array of characters (called a **string**), where the end of the string is indicated with the special character, `NULL` (whose value is 0).

We have also discussed the equivalence of subscripting expressions and pointer arithmetic; i.e. that a subscripting expression, `data[i]`, is equivalent to (and treated by the compiler as) the pointer expression, `*(data + i)`. Remember, the *name* of the array is a pointer variable, pointing to the first element of the array. These two forms of array access may be used interchangebly in programs, as fits the logic of the operation being performed. It is the semantics of pointer arithmetic that will compute the address of the indexed element correctly.

In addition, we have seen that passing arrays as parameters to functions is done by passing a pointer to the array (the array name). The cells of data are allocated in the calling function, and the called function can access them indirectly using either a pointer expression or a subscripting expression. Remember, a parameter like:

```
int func( int a[] )
```

(even if it has a <size> in the brackets) does NOT allocate integer cells for the array; it merely allocates an `int *` cell which will be given a pointer to an array in the function call. Such a parameter is exactly equivalent to:

```
int func( int *a)
```

We have discussed the fact that the pointer cell, referenced using the name of the array, is a *constant* pointer cell; i.e. it may not be changed by the program (it may not be used as an Lvalue). However; additional pointer cells of the appropriate type may be declared and initialized to point to the array (by the programmer) and can then be used to traverse the array with pointer arithmetic (such as the `++` or `--` operators).

We have shown how arrays can be initialized in the declaration (a bracketed, comma separated list of values, or, for strings, a string constant). We have seen the semantics of string assignment and how strings can be read and printed by `scanf()` and `printf()` using the `%s` conversion specifier. Remember, for `scanf()`, `%s` behaves like the numeric conversion specifiers; it skips leading white space and terminates the string (with a `NULL`) at the first following white space character.

Finally, we have shown an example of using arrays in a data base type applications, where arrays of different types were used to hold a collection of payroll records for individuals. In that example, the elements at a specific index in all of the arrays corresponded to one particular data record.

The *array* is an important and powerful data structure in any programming language. Once you master the use of arrays in C, the scale and scope of your programming abilities expand tremendously to include just about any application.

## 7.9 Exercises

With the following declaration:

```
int *p, x[10];
char *t, s[100];
```

Explain each of the following expressions. If there is an error, explain why it is an error.

1. (a) `x`
   (b) `x + i`
   (c) `*(x + i)`
   (d) `x++;`

2. (a) `p = x;`
   (b) `*p`
   (c) `p++;`
   (d) `p++;`
   (e) `p--;`
   (f) `--p;`

3. (a) `p = x + 5;`
   (b) `*p;`
   (c) `--p;`
   (d) `p*;`

4. `scanf("%s", s);`

   `Input: Hello, Hello.`

5. `printf("%s\n", s);`

6. 
```
scanf("%s", t);
t = s;
scanf("%s", t);
```

   Check the following problems; find and correct errors, if any. What will be the output in each case.

7.
```
main()
{   int i, x[10] = { 1, 2, 3, 4};

    for (i = 0; i < 10; i++) {
         printf("%d\n", *x);
         x++;
    }
}
```

8.
```
main()
{   int i, *ptr, x[10] = { 1, 2, 3, 4};

    for (i = 0; i < 10; i++) {
         printf("%d\n", *ptr);
         ptr++;
    }
}
```

9.
```
main()
{   int i, x[10] = { 1, 2, 3, 4};

    for (i = 0; i < 10; i++)
         printf("%d\n", (x + i));
}
```

10.
```
main()
{   int i, x[10] = { 1, 2, 3, 4};

    for (i = 0; i < 10; i++)
         printf("%d\n", *(x + i));
}
```

11.
```
main()
{   int i, *ptr, x[10] = {1, 2, 3, 4};

    ptr = x;
    for (i = 0; i < 10; i++) {
         printf("%d\n", *ptr);
         ptr++;
    }
}
```

12.
```
main()
{   int i, *ptr, x[10] = {1, 2, 3, 4};

    ptr = x;
    for (i = 0; i < 10; i++) {
         printf("%d\n", ptr);
```

```
            ptr++;
        }
    }
```

13. 
```
main()
{   char x[10];

    x = "Hawaii;
    printf("%s\n", x);
}
```

14. 
```
main()
{   char *ptr;

    ptr = "Hawaii";
    printf("%s\n", ptr);
}
```

15. 
```
main()
{   char *ptr, x[10] = "Hawaii";

    for (i = 0; i < 10; i++)
         printf("%d %d %d\n", x + i, *(x + i), x[i]);
}
```

16. 
```
main()
{   char x[10];

    scanf("%s", x);
    printf("%s\n", x);
}
```

The Input is:

```
Good Day to You
```

17. 
```
main()
{   char *ptr;

    scanf("%s", ptr);
    printf("%s\n", ptr);
}
```

The Input is:

```
Good Day to You
```

18. Here is the data stored in an array

```
char s[100];

Hawaii\0Manoa\0
```

What will be printed out by the following loop?

```
i = 0;
while (s[i]) {
     putchar(s[i]);
     i++;
}
```

## 7.10 Problems

1. Write a program that uses the sizeof operator to print the size of an array declared in the program. Use the sizeof operator on the name of the array.

2. Write a function that prints, using dereferenced pointers, the elements of an array of type float.

3. Write a function that checks if a given integer item is in a list. Traverse the array and check each element of the list. If an element is found return True; if the array is exhausted, return False.

4. Write a function that takes an array of integers and returns the index where the maximum value is found.

5. Write a function that takes an array and finds the index of the maximum and of the minimum.

   Use arrays to house sets of integers. A set is a list of items. Each item of a list is a member of a list and appears once and only once in a list. Write the following set functions.

6. Test if a number is a member of a set: is the number present in the set?

7. Union of two sets A and B: the union is a set that contains members of each of the two sets A and B.

8. Intersection of two sets A and B: the intersection contains only those members that are members of both the sets A and B.

9. Difference of two sets A and B: The new set contains elements that are members of A that are not also members of B.

10. Write a function to read a string from the standard input. Read characters until a newline is reached, discard the newline, and append a NULL. Use array indexing.

11. Write a function to read a string from the standard input. Read characters until a newline is reached, discard the newline, and append a NULL. Use pointers.

12. Write a function to write a string to the standard output. Write characters until a NULL is reached, discard the NULL, and append a newline. Use pointers.

13. Write a function to change characters in a string: change upper case to lower case and vice versa. Use array indexing.

14. Write a function to change characters in a string: change upper case to lower case and vice versa. Use pointers.

15. Write a function that counts and returns the number of characters in a string. Do not count the terminating NULL. Use array indexing.

16. Write a function that counts and returns the number of characters in a string. Do not count the terminating NULL. Use array indexing.

17. Write a function that removes the last character in a string. Use array indexing to reach the last element and replace it with a NULL.

18. Write a function that removes the last character in a string. Use pointers to traverse the array.

19. Repeat problems 24 and 25, but use the function of problems 22 or 23.