

Chapter Four

Array, String, and Pointer

Introduction

- Consider the following code that enters six numbers and outputs their sum:

```
sum = 0.0;
for (i = 0; i < 6; i++) {
    cin >> x;
    sum += x;
}
```

- This of course is easily extended to n values where n can be as large as required.
- However if it was required to access the values later the above would not be suitable.
- It would be possible to do it as follows by setting up six individual variables:

```
float a, b, c, d, e, f;
```

- and then handling each value individually as follows:

```
sum = 0.0;  
cin >> a;  
sum += a;  
cin >> b;  
sum += b;  
cin >> c;  
sum += c;  
cin >> d;  
sum += d;  
cin >> e;  
sum += e;  
cin >> f;  
sum += f;
```
- which is obviously a very tedious way to program

- If there were 10000 values imagine the tedium of typing the program (and making up variable names and remembering which is which)!
- To get round this difficulty all high-level programming languages use the concept of a data structure called an **Array**.

What is an array?

- is a data structure which allows a collective name to be given to a group of elements which ***all have the same type***.
- An individual element of an array is identified by its own unique ***index*** (or **subscript**).
- An array can be thought of as a collection of numbered boxes consecutively allocated and each containing one data item.
- The number associated with the box is the index of the item.
- To access a particular item the index of the box associated with the item is used to access the appropriate box.
- The index **must** be an integer and indicates the position of the element in the array.
- Thus the elements of an array are **ordered** by the index.

One Dimensional Array

- **Declaration of Arrays**

- **Syntax:**

- Type ArrayName [NumberOfElements];**

- Where **type** is the data type given for the elements of the array, **ArrayName** is a valid identifier for the array, **NumberOfElements** refers to the number of elements to be stored.
 - The number of elements **must be an integer**
 - For example:

- float salary [10];**
 - char name[20];**

- It is best to make the array size a constant and then, if required, the program can be changed to handle a different size of array by changing the value of the constant,

```
const int NE = 100;  
float salary[NE];
```

- **Accessing Array Elements**

- The **first element** in an array in C++ always has the **index 0**, and if the array has ***n*** elements the **last** element will have the **index *n*-1**.
- An **array element** is accessed by writing the identifier of the array followed by the subscript in square brackets.

```
ArrayName [index];
```

- For example:

```
salary [4] = 1500; //Assign 1500 to the 5th element in //  
the array
```

- Using an index, or subscript, that is out of range is called **Subscript Overflow**.
- Subscript overflow is one of the commonest causes of erroneous results and can frequently cause very strange and hard to spot errors in programs.
- **Initialization of arrays**
 - the initial values are given as a list enclosed in curly brackets.
 - For example:
`int primes[] = {1, 2, 3, 5, 7, 11, 13};`
 - Note that the array has not been given a size, the compiler will make it large enough to hold the number of elements in the list.
 - If the array is given a size then this size must be greater than or equal to the number of elements in the initialization list.
 - For example:
`int primes[10] = {1, 2, 3, 5, 7};`

- **Copying Arrays**

- The assignment operator *cannot* be applied to array variables:

```
const int SIZE=10
```

```
int x [SIZE] ;
```

```
int y [SIZE] ;
```

```
x = y ;           // Error – Illegal
```

- Only individual elements can be assigned to using the index operator,
 - For example: **x[1] = y[2];**
 - To copy the elements of array y into x a loop has to be used.

```
// Loop to do copying, one element at a time
```

```
for (int i = 0 ; i < SIZE; i++)
```

```
    x[i] = y[i];
```

Multidimensional arrays

- An array may have more than one dimension.
- Each dimension is represented as a subscript in the array.
- Therefore a two dimensional array has two subscripts, a three dimensional array has three subscripts, and so on.
- **Declaring Two Dimensional Array**
 - Syntax:
 - `DataType arrayName[NumberofRows][NumberofColumns];`
 - For example:

```
int board[8][8];
char sex[10][7];
```
- **Accessing Elements**
 - Use index for each dimension. Index starts from 0.
 - For example: `board[0][4]` -> refers to the first row and fifth column.

- **Initializing Multidimensional Arrays**

- For example:

```
int theArray[5][3] = { 1, 2, 3, 4, 5, 6,  
                      7, 8, 9, 10, 11, 12, 13, 14, 15};
```

- for the sake of clarity, the program could group the initializations with braces, as shown below.

```
int theArray[5][3] = {{1,2,3},{4,5,6},  
                     {7,8,9},{10,11,12},  
                     {13,14,15} };
```

- The compiler ignores the inner braces, which clarify how the numbers are distributed.

- **Omitting the Array Size**

```
int x[2][2] = { {1,2}, {3,4} } ;
```

OR

```
int x[][2] = { {1,2},{3,4} };
```

Strings Representation and Manipulation

- String in C++ is nothing but a sequence of character in which the last character is the **null** character '\0'.
- The null character indicates the end of the string.
- In C++ strings of characters are held as an **array of characters**, one character held in each array element.
- if a string has n characters then it requires an n+1 element array (at least) to store it.
- Thus the character 'a' is stored in a single byte, whereas the single-character string "a" is stored in two consecutive bytes holding the character 'a' and the null character.
- A string variable s1 could be declared as follows:
char s1[10];
 - The string variable s1 could hold strings of length up to nine characters since space is needed for the final null character.

- Strings can be initialized at the time of declaration just as other variables are initialized.
- For example:

```
char s1[] = "example";
```

```
char s2[20] = "another example"
```

- would store the two strings as follows:

```
s1 |e|x|a|m|p|l|e|\0|
```

```
S2 |a|n|o|t|h|e|r| |e|x|a|m|p|l|e|\0|?|?|?|?|
```

- **String Output**

- A string is output by sending it to an output stream,
- for example:

```
char s1[] = "example";
```

```
cout << "The string s1 is " << s1 << endl;
```

- The output will be:

```
The string s1 is example
```

- **String Input**

- When the input stream **cin** is used space characters, **newline** etc. are used as separators and terminators
- when inputting numeric data **cin** skips over any leading spaces and terminates reading a value when it finds a white-space character (**space**, **tab**, **newline** etc.).

- If the string **s1** was initialized as: **char s1[] = "example";** then the statement
cin >> s1;
- would set the string s1 as follows when the string **"first"** is entered (without the double quotes)
|f|i|r|s|t|\0|e|\0|
- Note that the last two elements are a relic of the initialization at declaration time.
- If the string that is entered is longer than the space available for it in the character array then C++ will just write over whatever space comes next in memory.
- This can cause some very strange errors when some of your other variables reside in that space!

- Reading a Word

- When you use the insertion operator `>>`, white spaces are considered as a terminating characters. It is used for a single word.

- For example:

```
char str [20];  
cout<<"Enter a string: ";  
cin>>str;  
cout<<"The string you typed is: ";
```

- Output:

```
Enter a string:  Law is a bottomless pit  
The string you typed is: Law
```

- It reads strings a single word, and throw away anything after the space.

- **Reading a Line - getline()**

- Syntax:

```
cin.getline(arrayName, size);
```

- For example:

```
const int max = 80;
```

```
char str[max];
```

```
cout<<"\n Enter a string;"
```

```
cin.getline(str, max); //max avoid buffer
```

```
//overflow
```

```
cout<<"\n You entered : "<<str;
```

- **Reading Multiple Lines - get()**
- Syntax:
`cin.get(arrayName, size, 'delimiter');`
- Where delimiter is any character that is used to tell the function to stop reading.
- The default value of this argument is the **newline('\n')** character.
- For example:

```
const int max = 80;  
char str[max];  
cout<<"\n Enter a string:\n";  
cin.get(str,max,'$'); //terminates with $  
cout<<\n You entered:\n"<<str;
```
- The function will continue to accept characters until you enter the terminated character \$
- Remember, you must still press Enter key after typing the '\$' character .

- **Avoiding buffer over flow**

- The strings in the program invites the user to type in a string.
- What happens if the user enters a string that is longer than the array used to hold it?
- There is no built-in mechanism in C++ to keep a program from inserting array elements outside an array.
- However, it is possible to tell the >> operator to limit the number of characters it places in an array.
- For example:

```
const int MAX=20;  
char str[MAX];  
cout<<"\n Enter a string: ";  
cin>>setw(MAX)>>str;  
cout<<"\n You entered :"<<str;
```

- **setw(width)** - requires to include a preprocessor called **iomanip**

- **Copying String**

- You can copy strings using **strcpy** or **strncpy** function
- We assign strings by using the string copy function **strcpy**.
- The prototype for this function is in **string.h**.
strcpy(destination, source);
- copies characters from the location specified by **source** to the location specified by **destination**.
- It stops copying characters after it copies the terminating null character.
- The return value is the value of the destination parameter.
- You must make sure that the destination string is large enough to hold all of the characters in the source string (including the terminating null character).

- Example:

```
#include <iostream.h>
#include <string.h>
void main(){
    char me[20] = "David";
    cout << me << endl;
    strcpy(me, "YouAreNotMe");
    cout << me << endl ;
    return;
}
```

- **strncpy**, copies only a specified number of characters.

strncpy(destination, source, int n);

- It may not copy the terminating null character.

- Example:

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[] = "String test";
    char str2[] = "Hello";
    char one[10];
    strncpy(one, str1, 9);
    one[9] = '\\0';
    cout << one << endl;
    strncpy(one, str2, 2);
    cout << one << endl;
    strcpy(one, str2);
    cout << one << endl;
}
```

- **Concatenating strings**

- use `strcat()` or `strncat`
- The function **`strcat`** concatenates (appends) one string to the end of another string.

`strcat(destination, source);`

- The first character of the source string is copied to the location of the terminating null character of the destination string.
- The destination string must have enough space to hold both strings and a terminating null character.
- **`strncat`** is like `strcat` except that it copies only a specified number of characters.

`strncat(destination, source, int n);`

- It may not copy the terminating null character.

- Example:

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[30];
    strcpy(str1, "abc");
    cout << str1 << endl;
    strcat(str1, "def");
    cout << str1 << endl;
    char str2[] = "xyz";
    strcat(str1, str2);
    cout << str1 << endl;
    str1[4] = '\0';
    cout << str1 << endl;
}
```

- Example:

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[30];
    strcpy(str1, "abc");
    cout << str1 << endl;
    strncat(str1, "def", 2);
    str1[5] = '\0';
    cout << str1 << endl;
    char str2[] = "xyz";
    strcat(str1, str2);
    cout << str1 << endl;
    str1[4] = '\0';
    cout << str1 << endl;
```

- **Comparing strings**

- Strings can be compared using **strcmp** or **strncmp** or **strcmpi** functions.
- The function **strcmp** compares two strings.

strcmp(str1, str2);

- **returns: < 0 if str1 is less than str2**
- **= 0 if str1 is equal to str2**
- **> 0 if str1 is greater than str2**

—

- Example:

```
#include <iostream.h>
#include <string.h>
void main() {
    cout << strcmp("abc", "def") << endl;
    cout << strcmp("def", "abc") << endl;
    cout << strcmp("abc", "abc") << endl;
    cout << strcmp("abc", "abcdef") << endl;
    cout << strcmp("abc", "ABC") << endl;
}
```

- **Strncmp** compares only a specified number of characters.

strncmp(str1, str2, int n);

- **strncmp** does not compare characters after a terminating null character has been found in one of the strings.

- **Example:**

```
#include <iostream.h>
#include <string.h>
void main()
{
    cout<< strncmp("abc","def",2) << "\n";
    cout<<strncmp("abc","abcdef",3)<<"\n";
    cout<<strncmp("abc","abcdef",2)<<"\n";
    cout<<strncmp("abc","abcdef",5)<<"\n";
    cout<<strncmp("abc","abcdef",20)<<"\n";
}
```

- **Strncmpi** compares two strings and ignore the case.
- For example:

```
#include <iostream.h>
#include <string.h>
void main(){
    char str1[]="Hello";
    char str2[]="hello";
    if(strncmpi(str1,str2)== 0)
        cout<<"They are equal\n";
    else if(strncmpi(str1,str2) < 0)
        cout<<str1<<" is less than "<<str2<<"\n";
    else
        cout<<str1<<" is greater than "<<str2<<"\n";
}
```

Pointers

- is simply the *address* of a memory location and provides an indirect way of accessing data in memory.
- A pointer variable is defined to 'point to' data of a specific type.
- Syntax:

Datatype *ptrVar;

- Example:

int *ptr1; // pointer to an int

char *ptr2; // pointer to a char

- The *value* of a pointer variable is the address to which it points.

- Operators
 - **&** -> the **address** operator
 - For example:

```
int *ptr1;  
int num = 9;  
ptr1 = &num;
```
 - it takes a variable as argument and returns the memory address of that variable.
 - The effect of the above assignment is that the address of num is assigned to ptr1.
 - ***** -> the **dereference** operator
 - it takes a pointer as argument and returns the contents of the location to which it points.
 - For example:

```
cout << *ptr1;
```
 - Prints the value of num which is 9.

- the type of a pointer must match the type of the data it is set to point to
- A pointer may be **cast** (type converted) to another type.
- For example:

```
int *ptr1;  
char *ptr2;  
ptr2 = (char*) ptr1;
```
- converts **ptr1** to **char** pointer before assigning it to **ptr2**.
- Regardless of its type, a pointer may be assigned the value **0** (called the **null** pointer).
- The **null** pointer is used for initializing pointers, and for marking the end of pointer-based data structures (e.g., linked lists).

```
#include <iostream>
using namespace std;
int main (){
    int firstvalue, secondvalue;
    int * mypointer;
    mypointer = &firstvalue;
    *mypointer = 10;
    mypointer = &secondvalue;
    *mypointer = 20;
    cout << "firstvalue is "
         << firstvalue << endl;
    cout << "secondvalue is "
         << secondvalue << endl;
    return 0;
}
```

Output:

```
firstvalue is 10
secondvalue is 20
```

```

#include <iostream>
using namespace std;
int main (){
int firstvalue = 5, secondvalue = 15;
int * p1, * p2;
p1 = &firstvalue; // p1 = address of firstvalue           p2 =
&secondvalue; // p2 = address of secondvalue
*p1 = 10; // value pointed by p1 = 10
*p2 = *p1; // value pointed by p2 = value pointed // by
p1
p1 = p2; // p1 = p2 (value of pointer is copied)
*p1 = 20; // value pointed by p1 = 20
cout << "firstvalue is " << firstvalue << endl;
cout << "secondvalue is " << secondvalue << endl;
return 0;
}
firstvalue is 10
secondvalue is 20

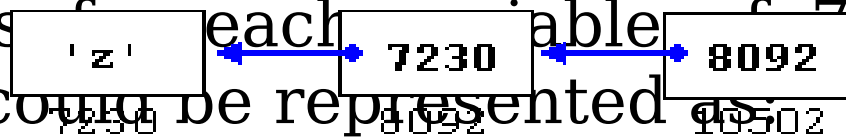
```

• Pointers to pointers

- C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers).
- In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations:

```
char a;  
char * b;  
char ** c;  
a = 'z';  
b = &a;  
c = &b;
```

- This, supposing the randomly chosen memory locations ^aeach ^bvariable ^c7230, 8092 and 10502, could be represented as



- In the above example which can be used in three different levels of indirection, each one of them would correspond to a different value:
 - **c** has type `char**` and a value of **8092**
 - ***c** has type `char*` and a value of **7230**
 - ****c** has type `char` and a value of **'z'**

- **Pointers and arrays**

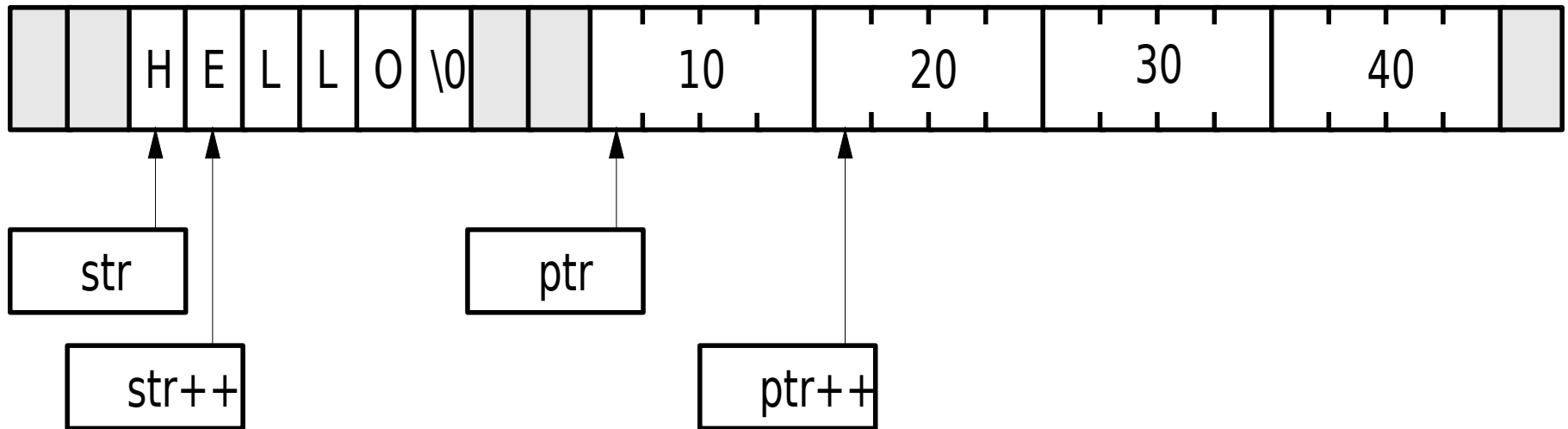
- The concept of array is very much bound to the one of pointer.
- the identifier of an array is equivalent to the address of its first element.
- For example, supposing these two declarations:
 - **int num[10];**
 - **int *p;**
- The following assignment operation would be valid:
 - **p = num;**
- However, the following allocation would not be valid:
 - **num = p;**
- Because an array can be considered a ***constant pointer***

- **Pointer Arithmetic**

- In C++ one can add an integer quantity to or subtract an integer quantity from a pointer.
 - This is frequently used by programmers and is called **pointer arithmetic**.
 - Pointer arithmetic is *not* the same as integer arithmetic,
- because the outcome depends on the size of the object pointed to.
 - For example, suppose that an **int** is represented by 4 bytes.

```
char *str = "HELLO";  
int  nums[] = {10, 20, 30, 40};  
int   *ptr = &nums[0]; // pointer to first  
                        //element
```

- **str++** advances **str** by one char (i.e., one byte) so that it points to the second character of "HELLO",
- whereas **ptr++** advances **ptr** by one int (i.e., four bytes) so that it points to the second element of **nums**



Pointer arithmetic.

- the elements of "HELLO" can be referred to as ***str**, ***(str + 1)**, ***(str + 2)**, etc.
- Similarly, the elements of **nums** can be referred to as ***ptr**, ***(ptr + 1)**, ***(ptr + 2)**, and ***(ptr + 3)**.

- Pointer arithmetic is very handy when processing the elements of an array.
- an array variable (such as **nums**) is itself the address of the first element of the array it represents.
- Hence the elements of **nums** can also be referred to using pointer arithmetic on **nums**, that is, **nums[i]** is equivalent to ***(nums + i)**
- The difference between **nums** and **ptr** is that:
 - **nums** is a constant, so it cannot be made to point to anything else,
 - whereas **ptr** is a variable and can be made to point to any other integer.

- Another form of pointer arithmetic allowed in C++ involves subtracting two pointers of the same type
- For example:

```
int *ptr1 = &nums[1];  
int *ptr2 = &nums[3];  
int n = ptr2 - ptr1; // n becomes 2
```

```
#include <iostream>
using namespace std;
int main (){
int numbers[5], * p;
p = numbers;
*p = 10;
p++;
*p = 20;
p = &numbers[2];
*p = 30;
p = numbers + 3;
*p = 40;
p = numbers;
*(p+4) = 50;
for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
return 0;
}
Output: 10, 20, 30, 40, 50,
```

- **Dynamic Memory**
 - **Heap** - is a memory area used for dynamically allocating memory blocks during program execution.
- it is also called **dynamic memory**
 - the program stack is also called **static memory**.
 - Two operators are used for allocating and deallocating memory blocks on the heap.
- **new** - takes a type as argument and allocated a memory block for an object of that type
 - It returns a pointer to the allocated block
 - For example:

```
int *ptr = new int;  
char *str = new char[10];
```
- allocate, respectively, a block for storing a single integer and a block large enough for storing an array of 10 characters.

- **delete** – is used for releasing memory blocks allocated by new.
 - It takes a pointer as argument and releases the memory block to which it points
 - For example:
 - delete ptr; // delete an object**
 - delete[] str; //delete an array of
//objects**
 - Note that when the block to be deleted is an array, an additional **[]** should be included to indicate this.