Module 6 Guidance Notes

Arrays & Strings

ENGG1340
Computer Programming II

COMP2113
Programming Technologies

Estimated Time of Completion: 3 Hours

Outline

There are 3 parts in this module:

I. (P. 3 – 47) Arrays – a basic data structure for storing a collection of objects of the same data type. You will also learn about performing some operations like searching and sorting of elements stored in an array.

II. (P. 48 – 67) Char & Char Arrays – here you will learn about some useful operations on char (one of the basic C/C++ data types), as well as the built-in string representation (which we called C-Strings) which is essentially some chars stored in an array.

III. (P. 68 – 93) C++ Strings – We will then go through the string class in C++ which provides you with a handful of functions for string operations.

Part I

ARRAYS

What are we going to learn?

- Array
- Passing array elements to functions
- Passing array to functions
- Searching / sorting an array
- Two dimensional arrays
- 2D array as function parameters
- char & char array

Handling Data of the Same Type

- Very often, a program needs to handle a collection of data of the same type
- Consider the following problem:
  - Write a program to input the scores of 80 students in a class and compute their average score and output those scores that are lower than the average.

```
int score_01, score_02, score_03, score_04, …, score_80;

cin >> score_01 >> score_02 >> … >> score_80;
double average = (score_01 + score_02 + … + score_80) / 80.0;

if (score_01 < average) cout << score_01 << endl;
if (score_02 < average) cout << score_02 << endl;
…
if (score_80 < average) cout << score_80 << endl;
```

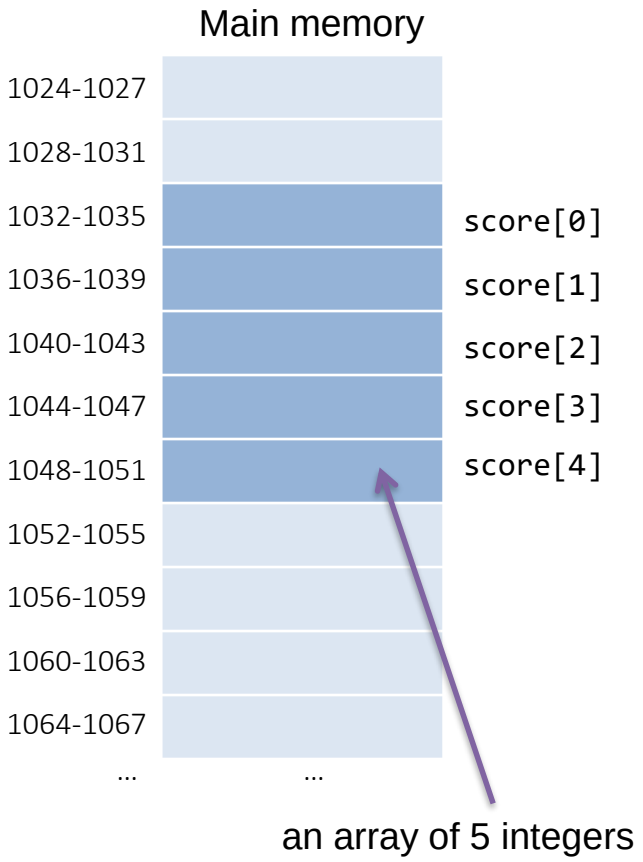
Using individually named variables to handle such data is cumbersome, especially for large datasets

Arrays

‡ Arrays in C++ provide a convenient way to process such data

± An array behaves like a list of variables (of the same type) with a uniform naming mechanism

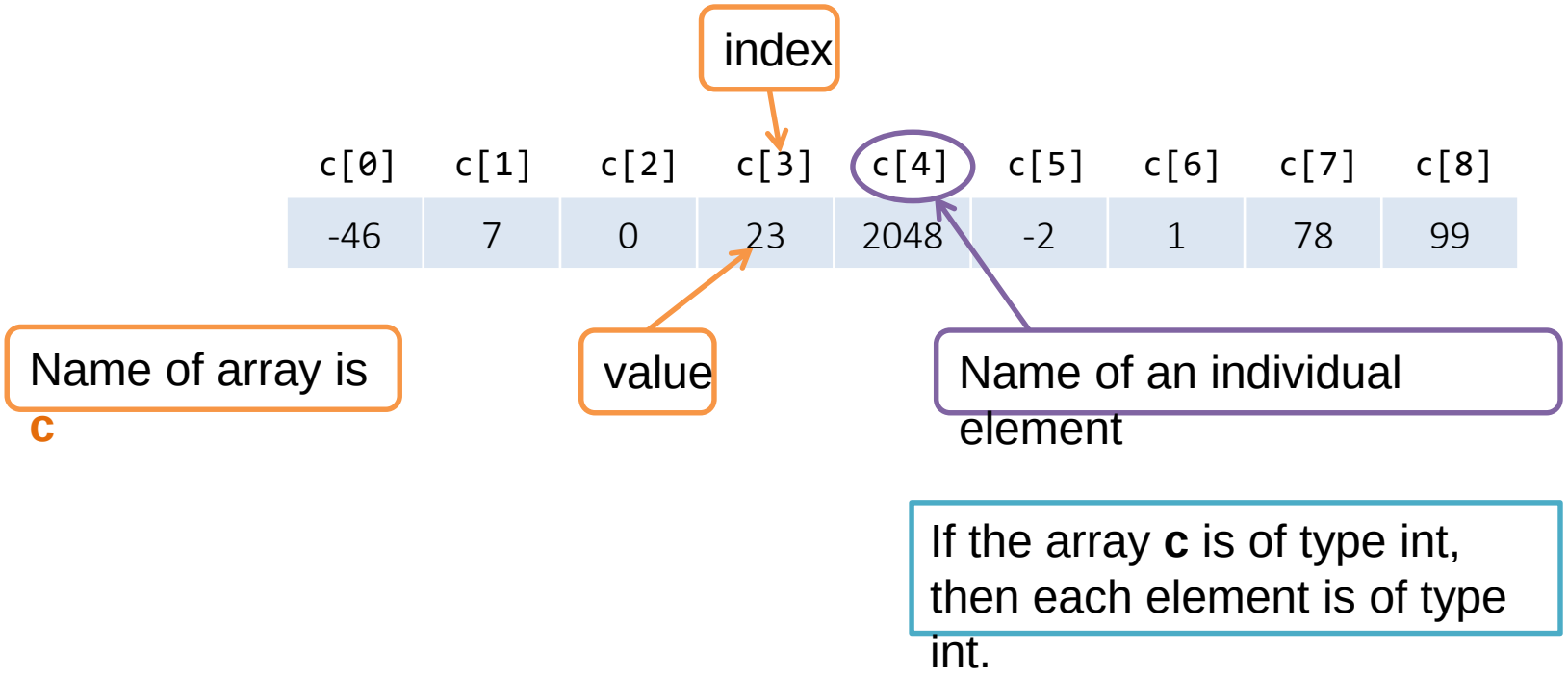
± An array is a consecutive group of memory locations that share the same type.



Note: Array is like list or tuple in Python. However, elements in Python lists or tuples can be of any data type. Elements in C++ arrays must be of the same data type.

Arrays

‡ Each element of an array can be regarded as a variable of the base type, and can be accessed by specifying the name of the array and the position (index) in the subscript operator [ ]



Indexes of Array Elements

‡ Array indexes always start from zero and end with the integer that is one less than the size of the array.

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]
-46	7	0	23	2048	-2

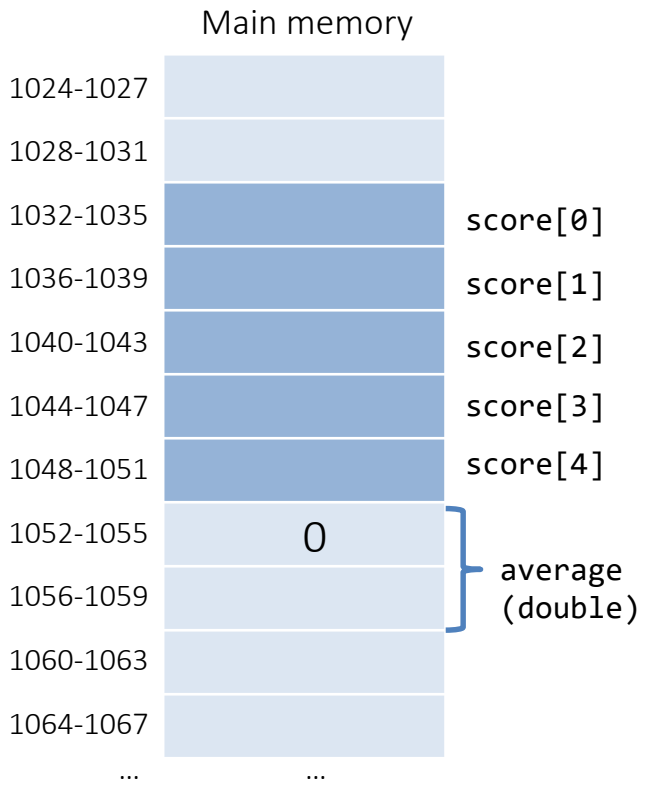
size of c is 6
elements are c[0], c[1], c[2], c[3], c[4], c[5]

‡ An array index can be any integer expression, including integer numerals and integer variables.

```
c[1] = 100;
cout << c[0] + c[1] + c[2] << endl;
int x = c[6] / 2;

int a = 1, b = 2;
c[a + b] += 2;      // c[3] = c[3] + 2

int i = 4;
c[i + 1] = c[i] - 30;    // c[5] = c[4] - 30
```

Indexes of Array Elements

- The compiler will NOT report any error when an array index that is out of range is used.
- On most systems, the program will proceed as if the index is legal and the memory cells corresponding to the nonexistent indexed variable will be accessed.
- This may unintentionally change the values of the memory cells probably belonging to some other variables.
- This is known as the array index out of bound error.



size of `score` is 5
what if we write
`score[5] = 0`?
Try in a program and see what happens

Declaring an Array

‡ An array declaration specifies the base type, the name and the size of the array.

Syntax

```
base_type    array_name[size];
```

‡ Arrays are **static** entities in that their sizes cannot be changed throughout program execution.

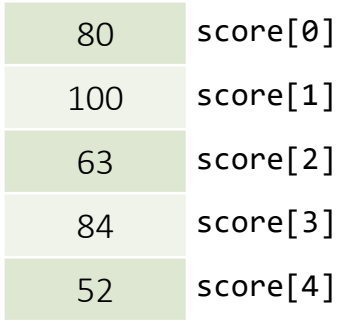
‡ Examples:

```
int score[5];           // an int array of 5 elements
char grade[8];          // a char array of 8 elements
double gpa[3];          // a double array of 3 elements

// Arrays and regular variables can be declared together
int max_score, min_score, score[5], passing_score;
```

Initialization with Initializer List

An array may be initialized in its declaration by using an equal sign followed by a list of values enclosed within a pair of braces { }.

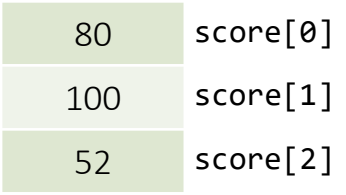
```
int score[5] = { 80, 100, 63, 84, 52 };
```



If an array is initialized in its declaration, the size of the array may be omitted and the array will automatically be declared to have the minimum size needed for the initialization values.

```
int score[] = { 80, 100, 52 };
```

size equals 3



Initialization with Initializer List

- The compiler will report an error if too many values are given in the initialization, e.g.,

```
int score[5] = {80, 100, 63, 84, 52, 96};
```

- It is, however, legal to provide fewer values than the number of elements in the initialization.
  - Those values will be used to initialize the first few elements.
  - The remaining elements will be initialized to a zero of the array base type.

```
int score[5] = {80, 100};
```

80	score[0]
100	score[1]
0	score[2]
0	score[3]
0	score[4]

How to initialize all elements to have value 0?

Initialization with Initializer List

- It is illegal to initialize or change the content of the whole array using an equal sign after its declaration.
- All the assignment statements below are therefore invalid.

```
int score[5];

score = { 80, 100, 63, 84, 52 };
score[] = { 80, 100, 63, 84, 52 };
score[5] = { 80, 100, 63, 84, 52 };
```

Example: Print the contents of an array with a loop

Need this library for setw()

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // use initializer list to initialize array n
    int n[10] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

    cout << "Element" << setw(13) << "Value" << endl;

    // output each array element's value
    for ( int j = 0; j < 10; ++j )
        cout << setw(7) << j << setw(13) << n[j] << endl;

    return 0;
}
```

Using a loop to access and print out each element

setw(): set the width (i.e., # of space) for the next item to be printed out

Initialization with a Loop

‡ Use a loop to access each element and initialize them to some initial values.

using the loop control variable **i** as the index

Using a loop to access and print out each element

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int n[10];  // n is an array 10 integers

    // initialize elements of array n to 0
    for ( int i = 0; i < 10; ++i )
        n[i] = 0;   // set element at location i to 0

    cout << "Element" << setw(13) << "Value" << endl;

    // output each array element's value
    for ( int j = 0; j < 10; ++j )
        cout << setw(7) << j << setw(13) << n[j] << endl;

    return 0;
}
```

Using an Array

Compare the following two implementations.

‡ Write a program to input the scores of 80 students in a class and compute their average score and output those scores that are lower than the average.

```
int score_01, score_02, score_03, score_04, …, score_80;

cin >> score_01 >> score_02 >> … >> score_80;
double average = (score_01 + score_02 + … + score_80) / 80.0;

if (score_01 < average) cout << score_01 << endl;
if (score_02 < average) cout << score_02 << endl;
…
if (score_80 < average) cout << score_80 << endl;
```

New version using array

```
int total = 0, score[80], i;
for (i = 0; i < 80; ++i)
{
    cin >> score[i];
    total += score[i];
}
double average = total / 80.0;
for (i = 0; i < 80; ++i)
    if (score[i] < average) cout << score[i] << endl;
```

Example 1

‡ To specify an array's size with a constant variable and to set array elements with calculations.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // constant variable can be used to specify array size
    const int arraySize = 10;

    int s[arraySize];   // array s has 10 elements

    for (int i = 0; i < arraySize; ++i)     // set the values
        s[i] = 2 + 2*i;

    cout << "Element" << setw(13) << "Value" << endl;

    // output contents of array s in tabular format
    for ( int j = 0; j < arraySize; ++j )
        cout << setw(7) << j << setw(13) << s[j] << endl;

    return 0;
}
```

Only need to change the value of `arraySize` to make the program scalable, i.e., for the program to work for other array sizes.

Example 2

‡ Using array elements as counters, e.g., roll a die and record the frequency of occurrences for each side.

‡ If `frequency[i]` stores the number of occurrences of face `i`, then what is the array size needed for storing the frequencies?

```
int  frequency[ 7 ];
// ignore element 0, use elements 1, 2, …, 6 only
```

‡ How to simulate a die-rolling?

Use a random number generator to generate a random number within [1..6] using the expression `rand() % 6 + 1`

Example 2

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    const int arraySize = 7;            // ignore element zero
    int frequency[arraySize] = {};      // initialize elements to 0

    srand( time(0) );   // seed random number generator

    // roll die 6,000,000 times; use die value as frequency index
    for (int roll = 1; roll <= 6000000; ++roll)
        ++frequency[ 1 + rand() % 6 ];

    cout << "Face" << setw(13) << "Frequency" << endl;

    // output each array element's value
    for ( int face = 1; face < arraySize; ++face )
        cout << setw(4) << face << setw(13) << frequency[face] << endl;

    return 0;
}
```

Optional Exercises

1. Write a program to initialize an array with the integers 1-10 and compute the sum of the 10 numbers.
2. Write a program to initialize an array with the first 10 odd integers starting from 1, and compute the product of the 10 numbers.
3. Write a program to initialize an array with the 10 characters ‘a’ to ‘j’ and print them out in reverse.
4. Write a program to get 10 input numbers from the users, print them out in reverse, and print out their sum.
5. Write a program to get input integers from the user repeatedly until the user enters 0. Your program should count the number of 1, 2, 3, 4, 5, 6 input by the user and print the frequencies out.

* Compare question 5 to the dice-rolling example in the previous slide.

Passing Array Elements to Functions

‡ Like regular variables, array elements can be passed to a function either by value or by reference.

```
// returns the square of an integer
int square( int x )
{
    return x * x;
}
```

Pass by value

To square each entry of an array

```
int a[4] = { 0, 1, 2, 3 };

for (int i = 0; i < 4; ++i)
{
   a[i] = square( a[i] );

}
```

What if this statement is replaced by this?

```
square( a[i] );
```

Passing Array Elements to Functions

‡ Like regular variables, array elements can be passed to a function either by value or by reference.

```
// returns the square of an integer
void square( int &x )
{
    x *= x;
}
```

Pass by reference

To square each entry of an array

```
int a[4] = { 0, 1, 2, 3 };

for (int i = 0; i < 4; ++i)
{
    square( a[i] );
}
```

Passing Arrays to Functions

‡ It is also possible to pass an entire array to a function (called an array parameter)

‡ To indicate that a formal parameter is an array parameter, a pair of square brackets [ ] is placed after its identifier in the function header and function declaration

Syntax (function header)

```
type_ret    func_name(base_type array_para[], ...)
```

Syntax (function declaration)

```
type_ret    func_name(base_type array_para[], ...);
```

Passing Arrays to Functions

‡ Examples

Function definition

```
void modifyArray( int b[], int arraySize )
{
    …
}
```

Function declaration (function prototype)

```
void modifyArray( int [], int);
```

Function call

```
int a[10];
modifyArray( a, 10);
```

Just need the array name here; no square brackets after the array identifier in function call

Passing Arrays to Functions

- An **array parameter** behaves very much like a pass-by-reference parameter.
  - The call functions can modify the element values in the callers' original arrays.
- An array argument only consists of the array identifier, but does not provide information of its size.
  - C++ **does not perform check** on the array bound, so we may pass an array of any size to a function.
  - **Another int argument** is often used to tell the function the size of the array.

Passing Arrays to Functions

```
int main()
{
    const int arraySize = 5; // size of array a
    int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a

    cout << "Effects of passing entire array:"
         << "\nThe values of the original array are:\n";

    // output original array elements
    for ( int i = 0; i < arraySize; ++i )
        cout << setw( 3 ) << a[ i ];

    cout << endl;

    // pass array a to modifyArray
    modifyArray( a, arraySize );
    cout << "The values of the modified array are:\n";

   // output modified array elements
    for ( int j = 0; j < arraySize; ++j )
        cout << setw( 3 ) << a[ j ];

    return 0;
}
```

See definition of modifyArray on the next slide

Passing Arrays to Functions

```
// in function modifyArray, "b" points to the
// original array "a" in memory
void modifyArray( int b[], int sizeOfArray )
{
    // multiply each array element by 2
    for ( int k = 0; k < sizeOfArray; ++k )
        b[ k ] *= 2;
}
```

```
Effects of passing entire array:
The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8
```

Screen output

* Note that the values of the array elements **are modified** by the function, which is of a similar effect as pass-by-reference

Searching an Array

‡ A common programming task is to search an array for a given value.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
-46	7	0	23	2048	-2	1	78	99

‡ Where is the item “78”? At index 7

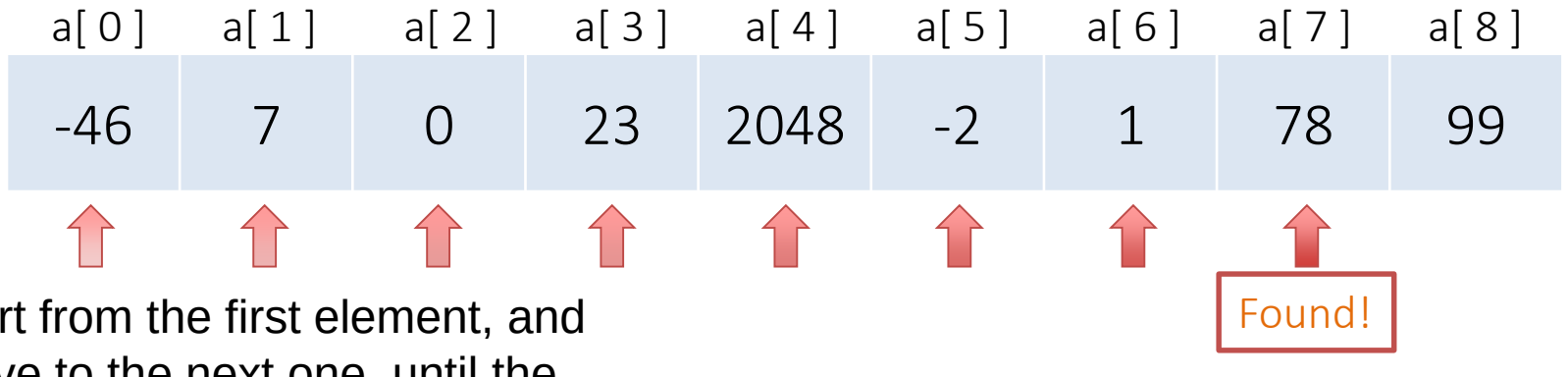
‡ Where is the item “100”? Not found

‡ If the value is found, the index of the array element containing the value is returned

‡ If the value is not found, –1 is returned

Linear Search

‡ The simplest method is to perform a linear search in which the array elements are examined sequentially from first to last



Start from the first element, and move to the next one, until the target item (78) is found.

How many elements need to be examined on average? Half of the array

How many elements need to be examined for the worst case? Entire array

Linear Search

```
// linear search of key value in array[]
// return the index of first occurrence of key in array[]
// return -1 if key is not found in array[]
int linearSearch( const int array[], int sizeOfArray, int key )
{
    for ( int j = 0; j < sizeOfArray; ++j )
        if ( array[ j ] == key ) // if found,
            return j;            // return location of key

    return -1;   // key not found
}
```

`const int array[]`: the `const` keyword is to specify that the contents of the formal parameter `array[]` are to remain constant (i.e., not to be changed) in this function.

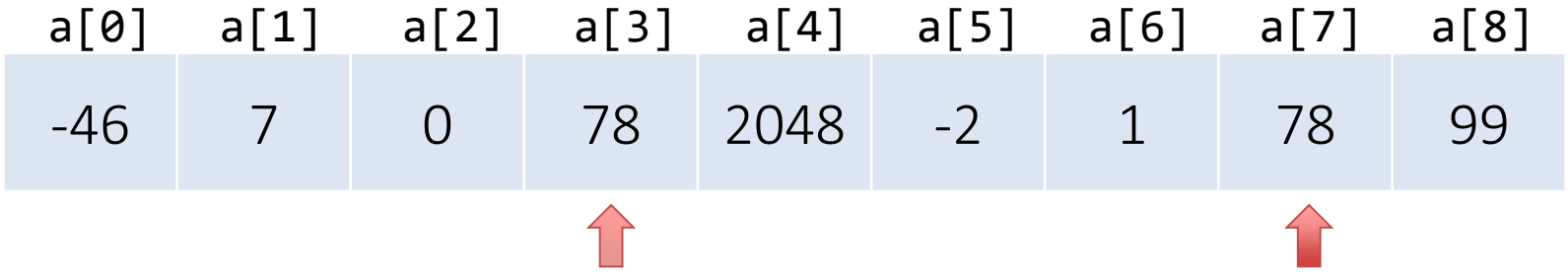
Linear Search

search.cpp

```
int main()
{
    const int arraySize = 10;  // size of array
    int a[ arraySize ];         // declare array a
    int searchKey;              // value to locate in array a

    // fill in some data to array
    for ( int i = 0; i < arraySize; ++i )
        a[i] = 2 * i;

    cout << "Enter an integer to search: ";
    cin >> searchKey;

    // try to locate searchKey in a
    int element = linearSearch( a, arraySize, searchKey );

    // display search results
    if ( element != -1 )
        cout << "Value found in element " << element << endl;
    else
        cout << "Value not found" << endl;

    return 0;
}
```

Linear Search (Variant)

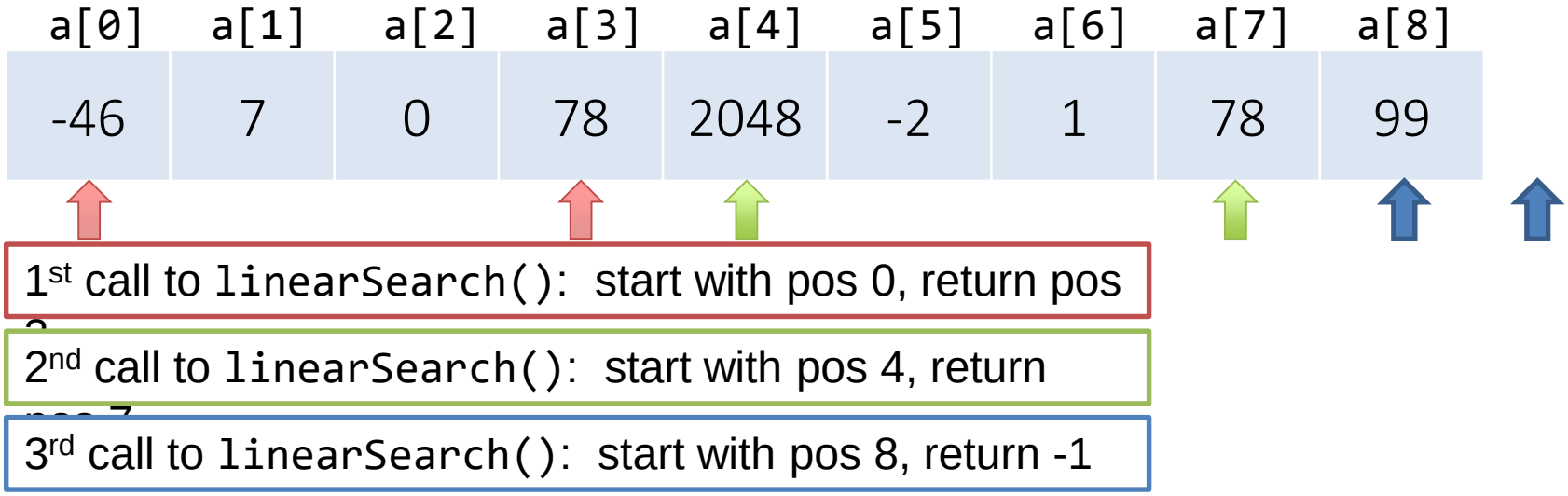
- The function `linearSearch()` returns only the first occurrence of the search item.
- What if we need the locations of ALL occurrences of the search item?



If search item = 78,
the program should be able to identify positions 3 and 7.

Linear Search (Variant)

- How to make changes to `linearSearch()` so that we can make use of it to look for all occurrences of an item?
- What does `linearSearch()` return?
- How about if we start searching from the returned position of a previous call of `linearSearch()`?



Linear Search (Variant)

‡ Function prototype for new `linearSearch()`

```
// linear search of key value in array[]
// starting search from startPos
// return the index of first occurrence of key in array[]
// return -1 if key is not found in array[]
int linearSearch( const int array[], int sizeOfArray,
                  int key, int startPos  );
```

‡ The `main()` function also needs some modification, so that `linearSearch()` will be called repeatedly until no more search item can be found.

Sorting an Array

- Another most widely encountered programming task is to sort the values in an array, e.g., in ascending/descending order.
- There are many different sorting algorithms, e.g., insertion sort, bubble sort, quicksort, etc.
- One of the easiest sorting algorithms is called selection sort.

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
before	-2	7	0	23	2048	-46
after sorting in ascending order	-46	-2	0	7	23	2048

Selection Sort

- A total of N iterations are needed to sort N elements
- At each iteration **i**, i = 0, …, N-1,
  - exchange **a[i]** with the **smallest** item among a[i]… a[N-1] (or the largest, if sort in descending order)

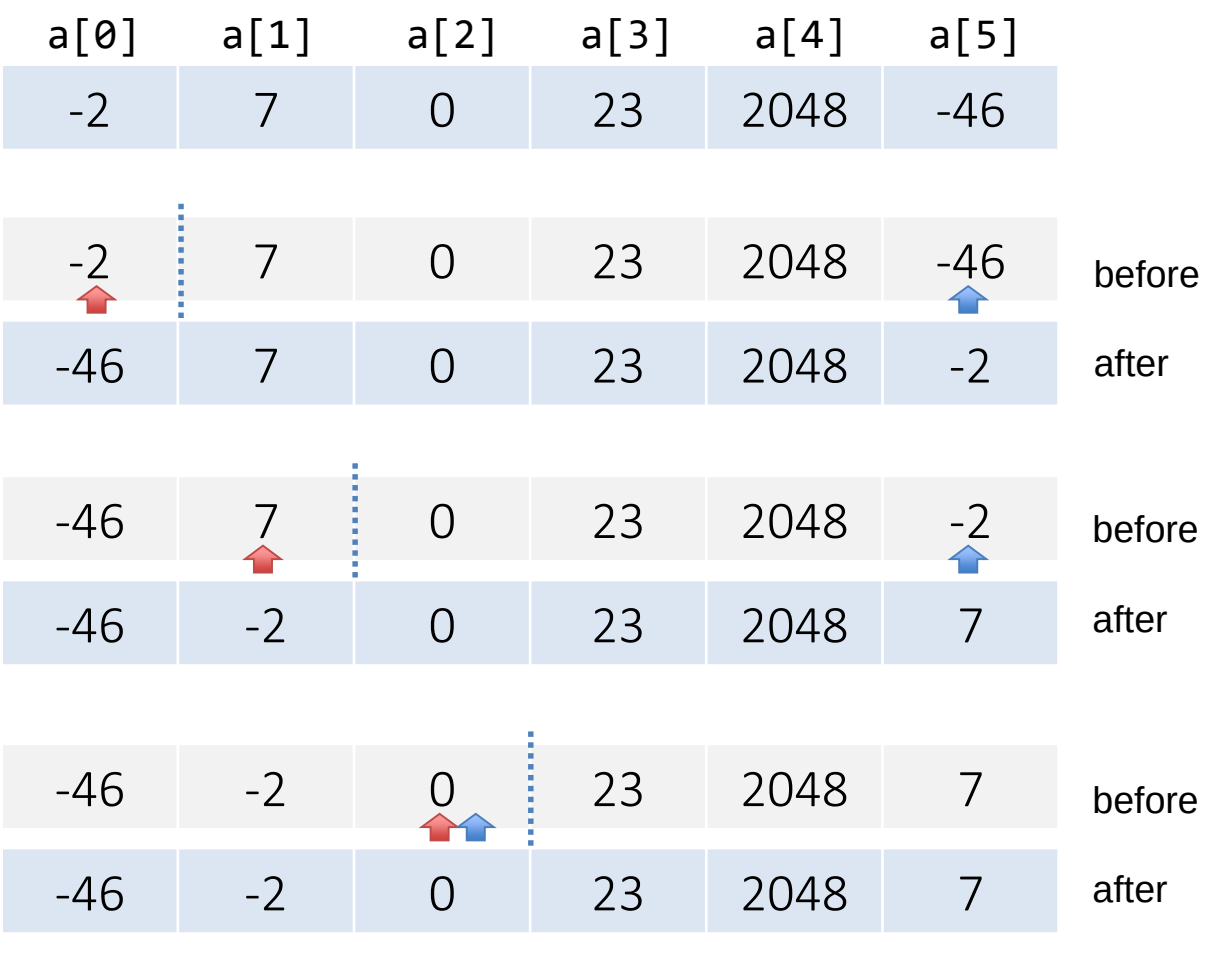
- An important property is that, after each iteration **i**,
  - the elements from a[0]…a[i] are sorted,
  - the elements from a[i+1]..a[N-1] remain to be sorted.

Selection Sort

: current element

: smallest element to the right of current item

To sort in ascending order

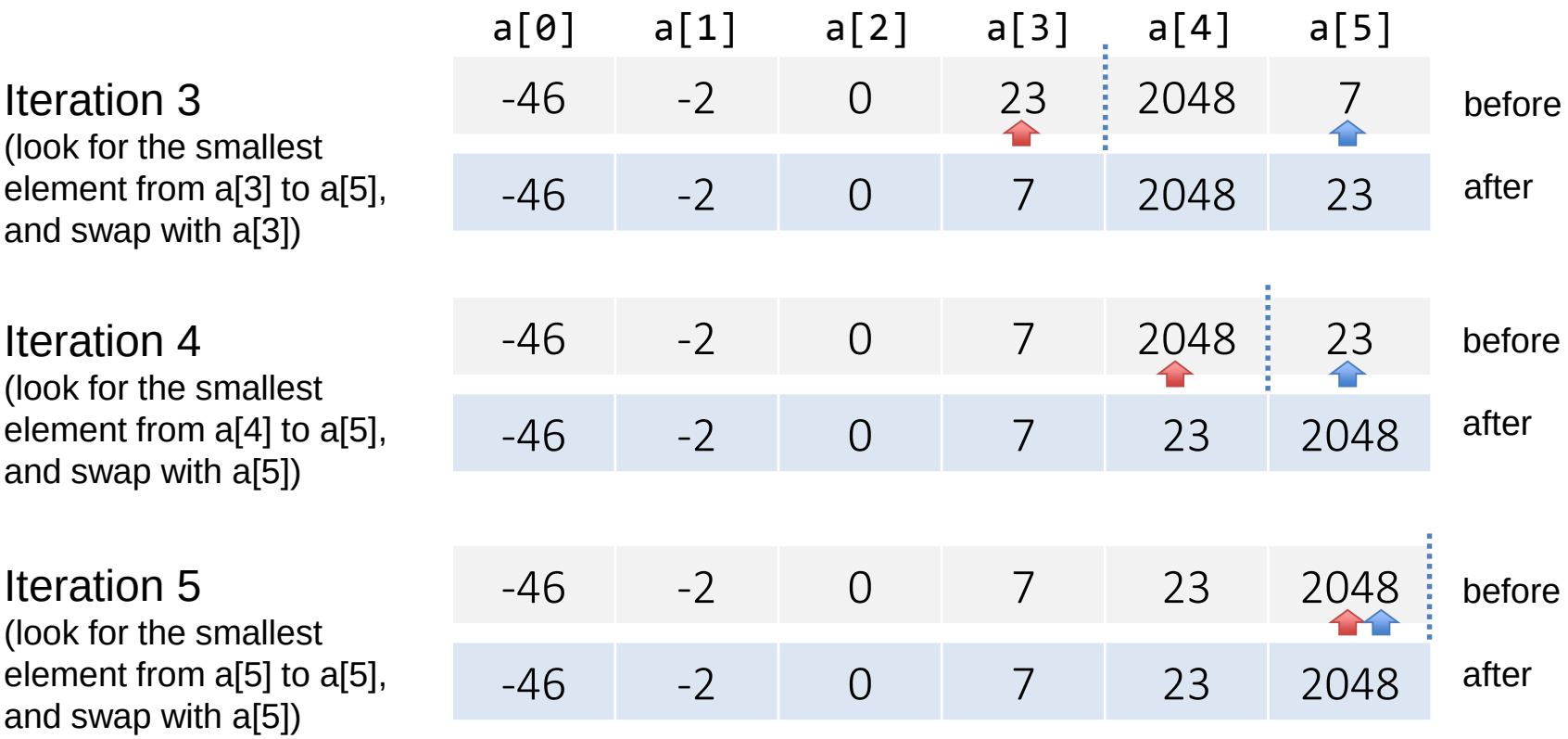


Selection Sort

: current element

: smallest element to the right of current item

To sort in ascending order



Selection Sort

```
// sort values in array[] in ascending order by selection sort
void sort(int array[], int sizeOfArray )
{
     int i, j, idx;
     int min;

     for ( i = 0; i < sizeOfArray; ++i )
     {
          min = array[i];
          idx = i;
          for ( j = i + 1; j < sizeOfArray; ++j )
          {
               if ( array[j] < min )
               {
                    min = array[j];
                    idx = j;
               }
          }
          if ( idx != i )
               swap( array[i], array[idx] );    // swap values
     }
}
```

```
void swap(int &a, int &b)
{
     int tmp = a;
     a = b;
     b = tmp;
     return;
}
```

Find the minimum from array[i] to array[N-1]

: array[i]

: array[idx]

Selection Sort

sort.cpp

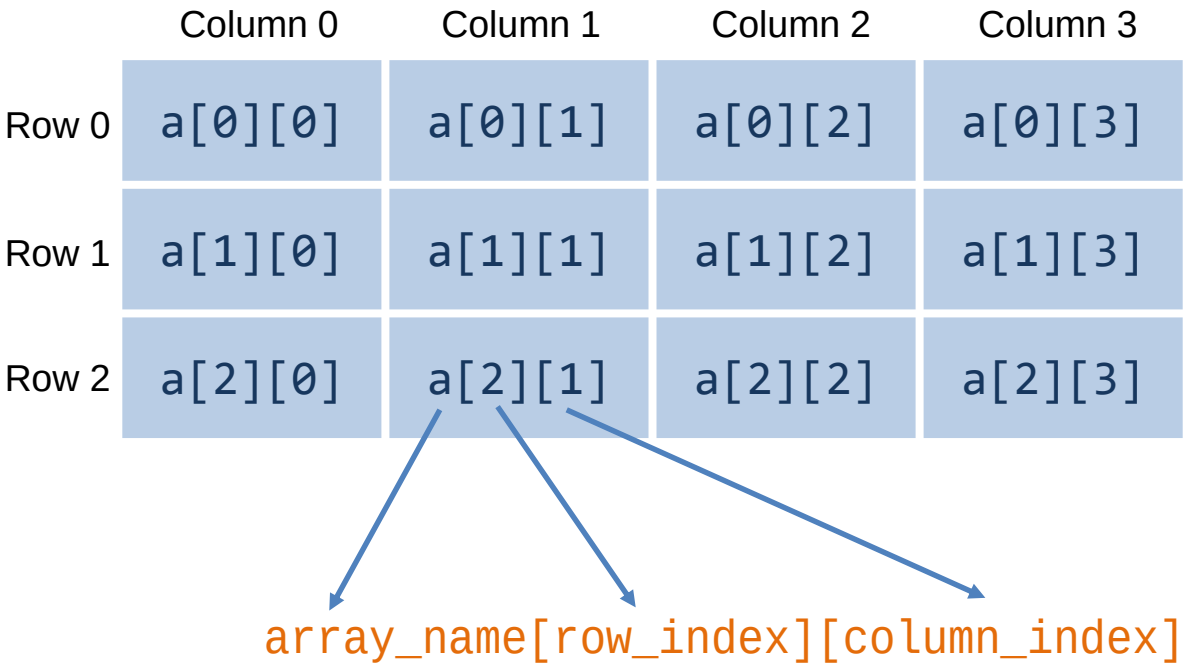
```
int main()
{
    const int arraySize = 6;                          // size of array
    int a[ arraySize ] = {-2, 7, 0, 23, 2048, -46};   // declare array a

    cout << "Original array: ";
    print_array( a, arraySize );

    sort( a, arraySize );

    cout << "Sorted array: ";
    print_array( a, arraySize );

    return 0;
}
```

```
void print_array( const int array[], int sizeOfArray )
{
    for ( int i = 0; i < sizeOfArray; ++i )
        cout << "[" << setw(2) << i << "] ";
    cout << endl;

    for ( int i = 0; i < sizeOfArray; ++i )
        cout << setw(3) << array[i] << "  ";
    cout << endl;
}
```

Two-Dimensional Arrays

‡ How about a table of values arranged in rows and columns?

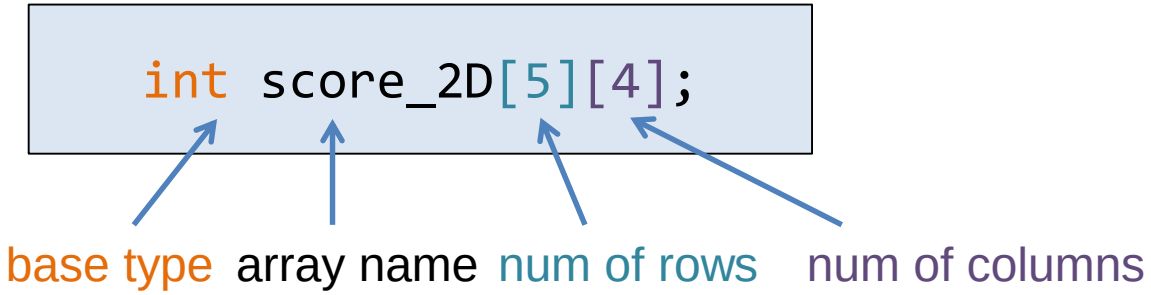
‡ A two-dimensional array (2D array):



A 2D array with
3 rows and
4 columns
(a **3-by-4 array**)

Two-Dimensional Arrays

‡ To declare a 2D array:



‡ Similar to the 1D case, each indexed variable of a multi-dimensional array is a variable of the base type, e.g.,

```
int score_2D[5][4];
score_2D[0][0] = 80;
score_2D[4][3] = score_2D[0][0] + 20;
```

Two-Dimensional Arrays

‡ Initialization:

```
int b[2][3] = { 1, 2, 3, 4, 5 };
```

fill up values for 1st row first, then 2nd row

b	0	1	2
0	1	2	3
1	4	5	0

Two-Dimensional Arrays

‡ Using a **nested for loop** to run through all elements.

```
const int nRows = 3;
const int nCols = 5;

int array2D[nRows][nCols];
int i, j;

// assign initial values
for (i = 0; i < nRows; ++i)
   for (j = 0; j < nCols; ++j)
         array2D[i][j] = nCols*i + j;
```

```
// print out array contents
for (i = 0; i < nRows; ++i)
{
   for (j = 0; j < nCols; ++j)
      cout << setw(3) << array2D[i][j] << ' ';
   cout << endl;       // start new line for each row
}
```

array2D.cpp

2D Array as Function Parameter

‡ Recall that for using a 1D array as parameter:

```
void print_1D_array ( int array [], int sizeOfArray );
```

indicate that this is an array of `int`

‡ When a 2D array parameter is used in a function header or function declaration, the size of the first dimension is not given, but the remaining dimension size must be given in square brackets.

‡ Now for using a 2D array as parameter:

```
void print_2D_array ( int array [][5], int numRows);
```

indicate that this is an array of `int[5]`

2D Array as Function Parameter

```
int main()
{
    const int nRows = 3;
    const int nCols = 5;

    int array2D[nRows][nCols];
    int i, j;

    // assign initial values
    for (i = 0; i < nRows; ++i)
        for (j = 0; j < nCols; ++j)
            array2D[i][j] = nCols*i + j;

    print_2d_array( array2D, nRows );

    return 0;
}
```

```
void print_2d_array( const int a[][5], int numRows)
{
    // print out array contents
    for (int i = 0; i < numRows; ++i) {
        for (int j = 0; j < 5; ++j)
            cout << setw(3) << a[i][j] << ' ';
        cout << endl;        // start new line for each row
    }
}
```

array2D_func.cpp

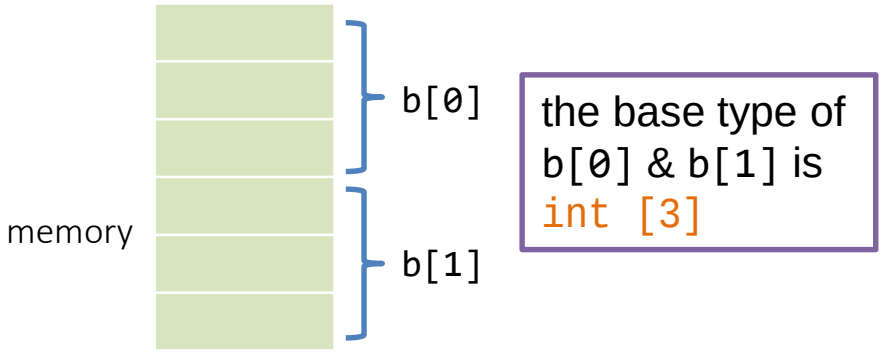
Reference Only

Multi-Dimensional Arrays

- Arrays with two or more dimensions are known as multi-dimensional arrays.

e.g.
```
int score_3D [5][4][3];
```

- A multi-dimensional array is an array of arrays.
  - All array elements are stored consecutively in memory, regardless of the number of dimensions.
  - E.g., `int b[2][3]` is a 1D array of size 2, with each element being a 1D integer array of size 3.



Part II

CHAR & CHAR ARRAYS

char Data Type

‡ Recall that the data type char is used for representing single characters, e.g., letters, digits, special symbols.

```
char c1 = 'a';     // the character 'a'
char c2 = '2';     // the character '2'
char c3 = '\n';    // the newline character
```

‡ Each char takes up **1 byte** of storage space.

‡ The most commonly used character set is ASCII (American Standard Code for Information Interchange), which uses 0-127 to represent a character.

The ASCII Character Set

Control Characters	
0	(Null character)
1	(Start of Header)
2	(Start of Text)
3	(End of Text)
4	(End of Trans.)
5	(Enquiry)
6	(Acknowledgement)
7	(Bell)
8	(Backspace)
9	(Horizontal Tab)
10	(Line feed)
11	(Vertical Tab)
12	(Form feed)
13	(Carriage return)
14	(Shift Out)
15	(Shift In)
16	(Data link escape)
17	(Device control 1)
18	(Device control 2)
19	(Device control 3)
20	(Device control 4)
21	(Negative acknowl.)
22	(Synchronous idle)
23	(End of trans. block)
24	(Cancel)
25	(End of medium)
26	(Substitute)
27	(Escape)
28	(File separator)
29	(Group separator)
30	(Record separator)
31	(Unit separator)
127	(Delete)

Printable Characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
4420.sG 0.00972 Tc[(103)] TJ ET EMC </MCID 121>> B0 720 540 re W* n BT /F2 9.984 Tf 1 0 0 1					

char and **int**

‡ Examples

```
char c = 'A';
cout << c << endl;
```

Screen output

```
A
```

```
char c = 65;
cout << c << endl;
```

Screen output

```
A
```

Since the data type of c is char, assigning an integer to c is treated as assigning an ASCII code to c.

char and **int**

‡ We may use an

char and **int**

‡ Arithmetic operations between char variables indeed operates on the ASCII values of the characters.

```
char letter1 = 'a';
char letter2 = 'b';
cout << letter1 << endl;
cout << letter2 << endl;

cout << letter1 – letter2 << endl;
cout << 'z' – 'a' << endl;

letter2--;
cout << letter2 << endl;
```

Screen output

```
a
b
-1
25
a
```

char and **int**

‡ More examples

```
char c = '1';
int num = c + 1;
cout << num << endl;
```

Screen output

```
50
```

The statement `int num = c + 1` takes the ASCII value of `'1'` (i.e., 49) for the addition operation.

```
char from = 'd';
char to = from – ('a' – 'A');
cout << to << endl;
```

Screen output

```
D
```

This is a technique to convert a small letter to its corresponding capital letter. The expression `'a' – 'A'` tells the difference in ASCII values between a small letter and its capital letter.

Comparisons for **char** Data Type

‡ How to determine if a letter is in lowercase or uppercase?

```
char letter;
cin >> letter;

if (    letter >= 'a' && letter <= 'z'    )
   cout << letter << " is in lowercase." << endl;
else if (   letter >= 'A' && letter <= 'Z'    )
   cout << letter << " is in uppercase." << endl;
```

Since the ASCII codes of the small letters and the capital letters are in order, we may use the relational operators (<, >, <=, >=) and equality operators (==, !=) to compare between characters.

Character Handling Functions

The <cctype> header file contains handy functions for character handling. Here are some examples:

`int isdigit(int c)`	Returns a nonzero (true) value if c is a digit, and 0 (false) otherwise
`int isalpha(int c)`	Returns a nonzero (true) value if c is a letter, and 0 (false) otherwise
`int isalnum(int c)`	Returns a nonzero (true) value if c is a digit or a letter, and 0 (false) otherwise
`int islower(int c)`	Returns a nonzero (true) value if c is a lowercase letter, and 0 (false) otherwise
`int isupper(int c)`	Returns a nonzero (true) value if c is an uppercase letter, and 0 (false) otherwise
`int tolower(int c)`	If c is an uppercase letter, returns c as lowercase letter. Otherwise, returns the argument unchanged.
`int toupper(int c)`	If c is a lowercase letter, returns c as an uppercase letter. Otherwise, returns the argument unchanged.

Reference only: check http://cplusplus.com/reference/cctype/ for more character handling functions

```
#include <iostream>
#include <cctype>

using namespace std;

int main()
{
  char a;

  a = '7';
  cout << a << (isdigit(a) ? " is " : " is not ") << "a digit" << endl;
  a = '$';
  cout << a << (isdigit(a) ? " is " : " is not ") << "a digit" << endl;

  a = 'B';
  cout << a << (isalpha(a) ? " is " : " is not ") << "a letter" << endl;
  a = 'b';
  cout << a << (isalpha(a) ? " is " : " is not ") << "a letter" << endl;
  a = '4';
  cout << a << (isalpha(a) ? " is " : " is not ") << "a letter" << endl;

  a = 'Z';
  cout << a << (islower(a) ? " is " : " is not ") << "a lowercase letter" << endl;
  a = 'z';
  cout << a << (islower(a) ? " is " : " is not ") << "a lowercase letter" << endl;
  a = '5';
  cout << a << (islower(a) ? " is " : " is not ") << "a lowercase letter" << endl;

  a = 'M';
  cout << a << (isupper(a) ? " is " : " is not ") << "an uppercase letter" << endl;
  a = 'm';
  cout << a << (isupper(a) ? " is " : " is not ") << "an uppercase letter" << endl;
  a = '#';
  cout << a << (isupper(a) ? " is " : " is not ") << "an uppercase letter" << endl;

  return 0;
}
```

charfunc.cpp

We will rewrite this program in C in part IV later.

Character Handling Functions

Screen output of charfunc.cpp

```
7 is a digit
$ is not a digit
B is a letter
b is a letter
4 is not a letter
Z is not a lowercase letter
z is a lowercase letter
5 is not a lowercase letter
M is not an uppercase letter
m is an uppercase letter
# is not an uppercase letter
```

Text as Strings

We talked about characters.

How about if we want to represent a sequence of characters?

```
cout << "Hello World!" << endl;
```

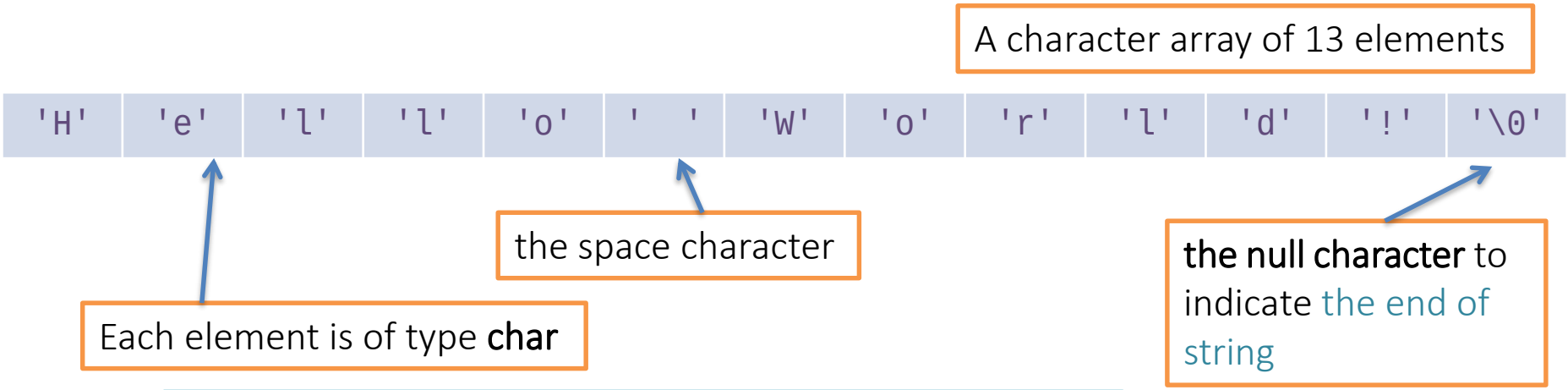
Strings are a sequence of characters and in C++ we use a pair of double quotation marks to enclose a string.

```
"Hello World!"
“COMP2113"
"@_@"
```

C-Strings (Character Arrays)

The low-level internal representation in C/C++ of a string (i.e., how a string is stored in memory) is an **array of char** (i.e., a character array), which is ended by a **null character** ('\0'). We call this a **C-String** or a null-terminated string.

The internal representation of the string **"Hello World!"**



C-Strings (Character Arrays)

‡ What is the difference between `'A'` and `"A"`?

`'A'`

a char **‘A’** is represented internally (i.e., in memory) using one byte

`'A'` `'\0'`

a string **"A"** is represented internally using two bytes **'A'** and **'\0'**

‡ What is the difference between `'0'` and `'\0'`?

`'0'`

the byte value of this char is 48, representing the digit 0

`'\0'`

the byte value of this char is 0, representing the null character

Also refer to the ASCII table on this page.

C-Strings (Character Arrays)

‡ Declaring a character array and assign a string to it:

```
char name[16] = { 'J', 'o', 'h', 'n', '\0'};
```

‡ Examples:

```
char name[16] = { 'J', 'o', 'h', 'n', '\0'};
cout << name;
```

```
John
```

Screen output

You can see that C++ treats the character array name[] as a string

C-Strings (Character Arrays)

Or you can simply do the followings to declare a C-string:

```
char name[16] = "John";
```

1

```
char name[] = "John";
```

2

What's the difference between the above two declarations?

In 1, the size of the array **name** is of 16 chars;
and in 2, the size is of 5 chars (i.e., C/C++ automatically determines the array size in this case.)

Q: Why is the size 5 chars in case 2?

C-Strings (Character Arrays)

‡ Like regular arrays, it is not possible to copy blocks of data to a character array using an equal sign (i.e., an assignment) after its declaration.

‡ Hence, all the assignment statements below are invalid.

```
char name[16];

name = { 'J', 'o', 'h', 'n', '\0' };
name[] = { 'J', 'o', 'h', 'n', '\0' };

name = "John";
name[] = "John";
```

C-Strings (Character Arrays)

We may access each individual character using the subscript operator [], just as for an ordinary array.

```
char name[] = "Steve";
cout << name << endl;

name[2] = 'o';
cout << name << endl;
```

Screen output

```
Steve
Stove
```

The Null character

Recall that the null character '\0' is to indicate end of string.

What is the output of the following program segment?

```
char name[] = "Steve";
cout << name << endl;

name[5] = 'n';
cout << name << endl;
```

Screen output

```
Steve
Steven??@#v
```

Note that here we overwrite the null character `'\0'` at `name[5]` with `'n'`, so what will be the output of the cout statement?

Since the null character is overwritten, we have an unexpected end of string. 7 K H µ J D U E D J H ¶ E \ W H F R memory that follows the array memory will just be printed out, as if they constitute part of the string.

In this particular example, the size of name[] is 6. The cout statement will also risk an index-out-of-bound error.

Working with C-Strings

‡ cout and cin can be used for I/O for C-strings:

```
char msg[] = "Please enter your name: ";
char name[80];

cout << msg;
cin >> name << endl;

cout << "Hello " << name << "!" << endl;
```

```
Please enter your name:
Steve
Hello Steve!
```

Screen output

Side-notes only: The <cstring> header in C++ provides a set of functions for C-string manipulation, e.g., string copy **strcpy()**, string compare **strcmp()**, string length **strlen()**. See http://cplusplus.com/reference/cstring/ for details.

Part III

C++ STRINGS

What are we going to learn?

- The `string` class
- String concatenation
- String comparison
- String I/O
- Member functions of the string class for string manipulation, e.g.,
  - `string::length()`
  - `string::empty()`
  - `string::substr()`
  - `string::find()`
  - `string::rfind()`
  - …

The **string** Class

‡ Handling C-string is rather low level, e.g., one will need to deal with the internal representation (i.e., the character array) and the null characters.

‡ C++ standard library provides a class (i.e., programmer defined data type) named `string` for more convenient handling of strings.

‡ You may think of C++ string as a wrapper/container for handling char arrays. It provides handy string operations so basically you don't need to care about its underlying representation.

String Initialization

‡ We need to include the header file **<string>** to use the class **string**:

```
#include <string>
```

‡ A string object can be declared using the class name string and initialized with a C-string or another string object:

```
char a[80] = "Hello";     // a C-string
string msg1 = a;          // initialized with a C-string
string msg2 = "World";    // initialized with a string literal
string msg3 = msg1;       // initialized with a string object
```

String Assignment

‡ The string class has its own end-of-string representation, for which we do not need to handle.

‡ **Unlike C-string**, we can initialize or change a string object using an assignment statement after its declaration:

```
char a[80] = "Hello";     // C-string declaration
string msg1, msg2, msg3;  // string declarations

msg1 = a;          // initialized with a C-string
msg2 = "World";    // initialized with a string literal
msg3 = msg1;       // initialized with a string object
```

Compare the above with C-string declaration on this slide

String ±Subscript Operator

‡ We may also access individual character using the subscript operator [ ]:

```
string msg = "Hello World!";
msg[11] = '?';

cout << msg << endl;
```

```
Hello World?
```

Screen output

String Concatenation

‡ Two strings can be concatenated to form a longer string using the binary operator +

```
string msg1 = "I love ";
string msg2 = "cats";
string msg3 = msg1 + msg2;
string msg4 = msg1 + "dogs";
string msg5 = "I hate " + msg2 + " and dogs";
```

I love cats

I love dogs

I hate cats and dogs

‡ Note that at least one of the operands of + must be a string object.

Here, both operands are string literals (i.e., constants)

```
string msg = "I love " + "dinosaurs";
```

String Comparison

‡ Strings can be compared lexicographically (dictionary order) using relational (>, <, >=, <=) and equality (==, !=) operators. The comparison is carried out in a character by character manner from left to right.

```
string msg1 = "Apple", msg2  = "apple";
string msg3 = "apples", msg4 = "orange";

bool c1 = msg1 == msg2;
bool c2 = msg1 < msg2;
bool c3 = msg2 < msg3;
bool c4 = msg3 != msg4;
bool c5 = msg4 > msg3;
```

Note: at least one of the operands need to be a string object

c1	false	c2	true	c3	true	c4	true	c5	true

I/O with String Objects

- Both `cout` and `cin` support string objects.
- The insertion operator `<<` and extraction operator `>>` work the same for string objects as for other basic data types

```
string msg;
cin >> msg;
cout << msg;
```

- Note that:
  - The extraction operator >> ignores whitespace at the beginning of input and stops reading when it encounters more whitespaces.
  - The word received by a string object will therefore have any leading and trailing whitespace deleted.
  - Cannot read in a line or string that contains one or more blanks

I/O with String Objects

Example

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string word1, word2, word3;
   cout << "Please input a sentence: " << endl;

   cin >> word1 >> word2 >> word3;

   cout << "Word 1 = \"" << word1 << "\"\n"
        << "Word 2 = \"" << word2 << "\"\n"
        << "Word 3 = \"" << word3 << "\"\n";
   return 0;
}
```

```
Please input a sentence:
I   love       dogs
Word 1 = "I"
Word 2 = "love"
Word 3 = "dogs"
```

Screen output

Use \" for a " character in a string

string_io.cpp

How do we read in an entire line including spaces from the input then?

Reading a Line from Input

‡ We use the library function `getline()` to read in a line from the standard input and store the line in a string:

```
string s;
cout << "Please input a sentence: " << endl;
getline(cin, s);
cout << "s = \"" << s << "\"\n";
```

string_getline.cpp

```
Please input a sentence:
I   love        dogs
s = "I   love        dogs"
```

Screen outputs

Reading a Line from Input

‡ The function `getline()` can be used to read in a line from the current position until a **delimitation character** is encountered

```
string s;
cout << "Input 2 comma-separated phrases: " << endl;
getline(cin, s, ',');
cout << "1st phrase = \"" << s << "\"\n";
```

string_getline.cpp

```
Input 2 comma-separated phrases:
Stay hungry, stay foolish
1st phrase = "Stay hungry"
```

Screen outputs

As you can see, without providing the third D U J X P H Q W µ ¶ L Q \ case), the default delimitation character for the getline function is the newline character \µQ ¶

Member Functions

- The class string has a number of member functions which facilitate string manipulation, which includes
  - `string::length()` – returns length of the string
  - `string::empty()` – returns whether the string is empty
  - `string::substr()` – returns a substring
  - `string::find()` – finds the first occurrence of content in the string
  - `string::rfind()` – finds the last occurrence of content in the string
  - `string::insert()` – inserts content into the string
  - `string::erase()` – erases characters from the string
  - `string::replace()` – replaces part of the string

More member functions can be found at http://www.cplusplus.com/reference/string/string, but you are expected to be get familiar with the above functions only for this course.

string::length()

‡ Returns the number of characters in a string object

```
string s = "Stay hungry, stay foolish";
int n = s.length();
cout << "s has " << n << " characters. " << endl;
```

Note we use . to invoke the member function of a string object.
E.g., here s.length() means that we call the string::length() member function of the string object s.

```
s has 25 characters.
```

Screen outputs

string::empty()

‡ Returns true if a string object is empty; false otherwise

```
string s;
if (s.empty())
  cout << "s is empty." << endl;
else
  cout << "s has " << s.length() << " characters.\n";
```

```
s is empty.
```

Screen outputs

What if `s = " "`?
Is this an empty string?

No, this is a string with a space character, and its length is 1.

string::erase()

- Erase `n` characters starting at a specific position `pos` from the current string
- Note that the string will be modified

Example

```
string firstName = "Alan";
string name = firstName + " Turing";
string str1 = "It is sunny. ";
string str2 = "";
string str3 = "C++ programming.";
string str4 = firstName + " is taking " + str3;

cout << str1.empty() << endl;
cout << str2.empty() << endl;
str3.erase(11,4);
cout << str3 << endl;
cout << firstName.length() << endl;
cout << name.length() << endl;
cout << str4 << endl;
```

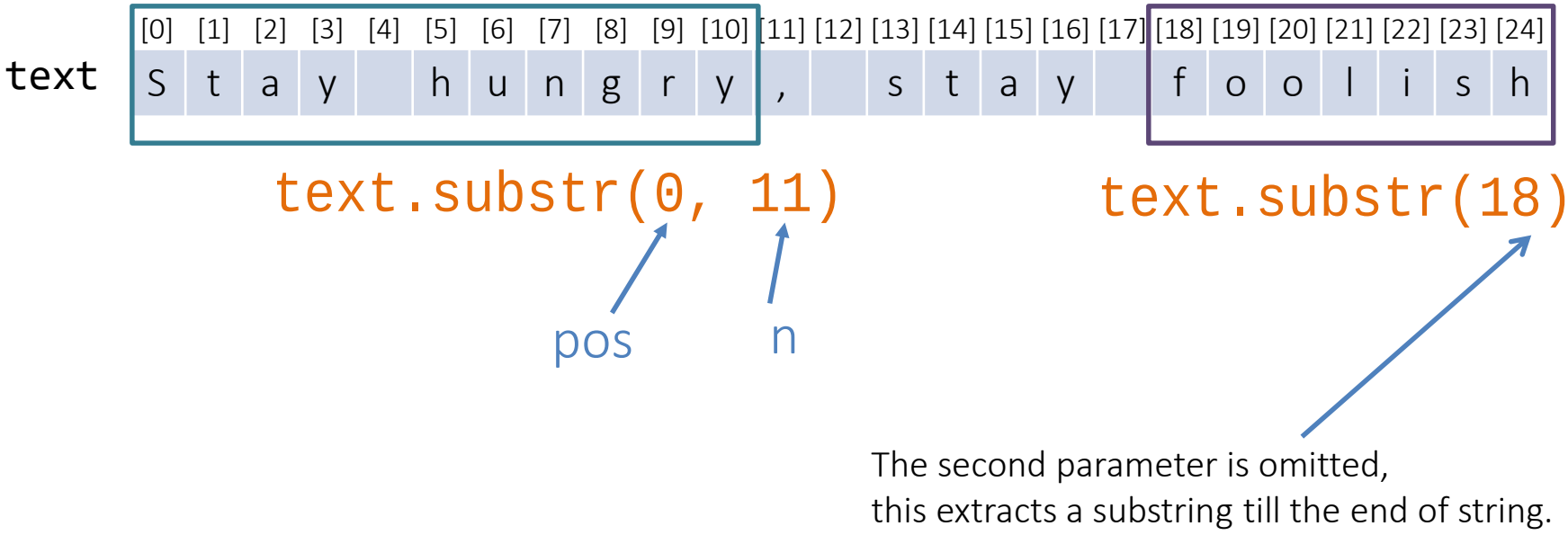
string_op.cpp

```
0
1
C++ program.
4
11
Alan is taking C++ programming.
```

Screen outputs

string::substr()

‡ Returns a substring of the current string object starting at the character position pos and having a length of n characters



string::substr()

Example

```
string s;
string str;

s = "It is cloudy and warm.";

cout << s.substr(0, 5) << endl;
cout << s.substr(6, 6) << endl;
cout << s.substr(6, 16) << endl;
cout << s.substr(17, 10) << endl;
cout << s.substr(3, 6) << endl;
str = s.substr(0, 8);
cout << str << endl;
str = s.substr(2, 10);
cout << str << endl;
```

Screen outputs

```
It is
cloudy
cloudy and warm.
warm.
is clo
It is cl
 is cloudy
```

substring.cpp

string::find()

‡ Searches a string object for a given string `str`, and returns the position of the first occurrence

```
find(str)
```

‡ When `pos` is specified the search only includes characters at or after position `pos`, ignoring any possible occurrences in previous locations.

```
find(str, pos)
```

‡ If there is no occurrence of `str`, the constant value `string::npos` (i.e., –1) will be returned.

string::find()

Example

```
string s = "Outside it is cloudy and warm.";
string t = "cloudy";

cout << s.find("is") << endl;
cout << s.find('s') << endl;
cout << s.find(t) << endl;
cout << s.find('i', 6) << endl;
cout << s.find('o') << endl;
if (s.find("the") == -1)
   cout << "not found" << endl;
if (s.find("the") == string::npos)
   cout << "not found" << endl;
```

string_find.cpp

This example shows that the search is case-sensitive

Screen outputs

```
11
3
14
8
16
not found
not found
```

string::rfind()

‡ Searches the current string object for the content specified in `str`, and returns the position of the last occurrence

```
rfind(str)
```

This is essentially to search in the reverse direction from the end of the string

‡ When `pos` is specified the search only includes characters at or before position `pos`, ignoring any possible occurrences in later locations.

```
rfind(str, pos)
```

‡ If there is no occurrence of str, the constant value `string::npos` (i.e., –1) will be returned.

string::rfind()

Example

```
string s = "Outside it is cloudy and warm.";
string t = "cloudy";

cout << s.rfind("is") << endl;
cout << s.rfind('s') << endl;
cout << s.rfind(t) << endl;
cout << s.rfind('i', 6) << endl;
cout << s.rfind('o') << endl;
if (s.rfind("the") == -1)
  cout << "not found" << endl;
if (s.rfind("the") == string::npos)
  cout << "not found" << endl;
```

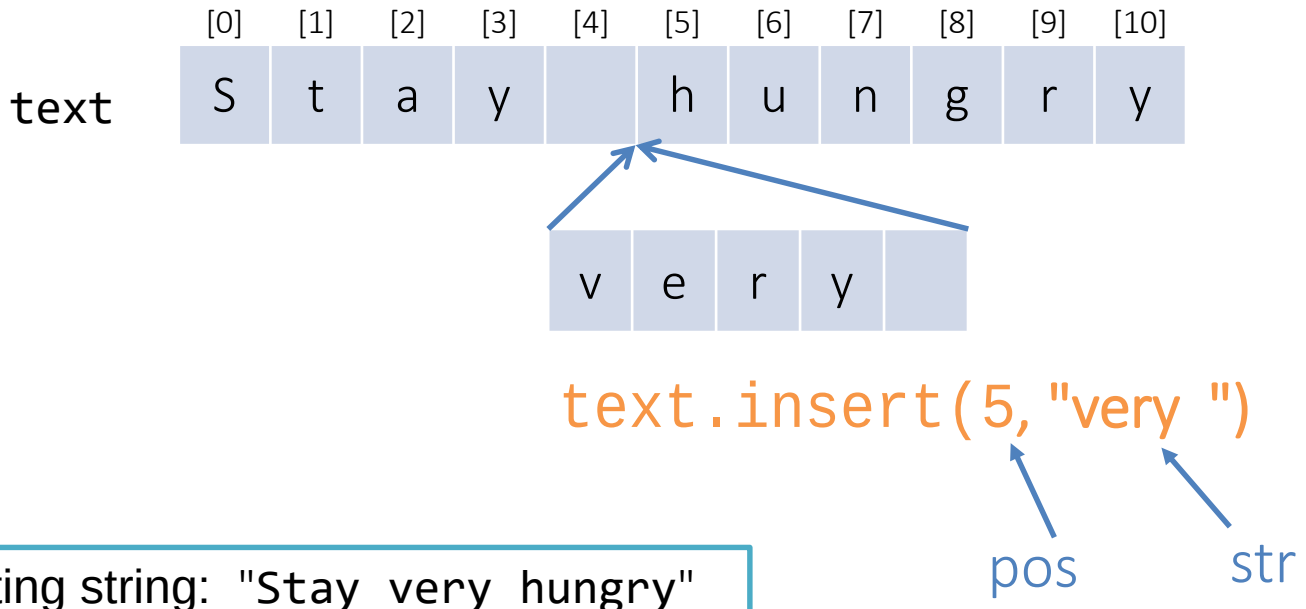
string_rfind.cpp

Screen outputs

```
11
12
14
4
16
not found
not found
```

string::insert()

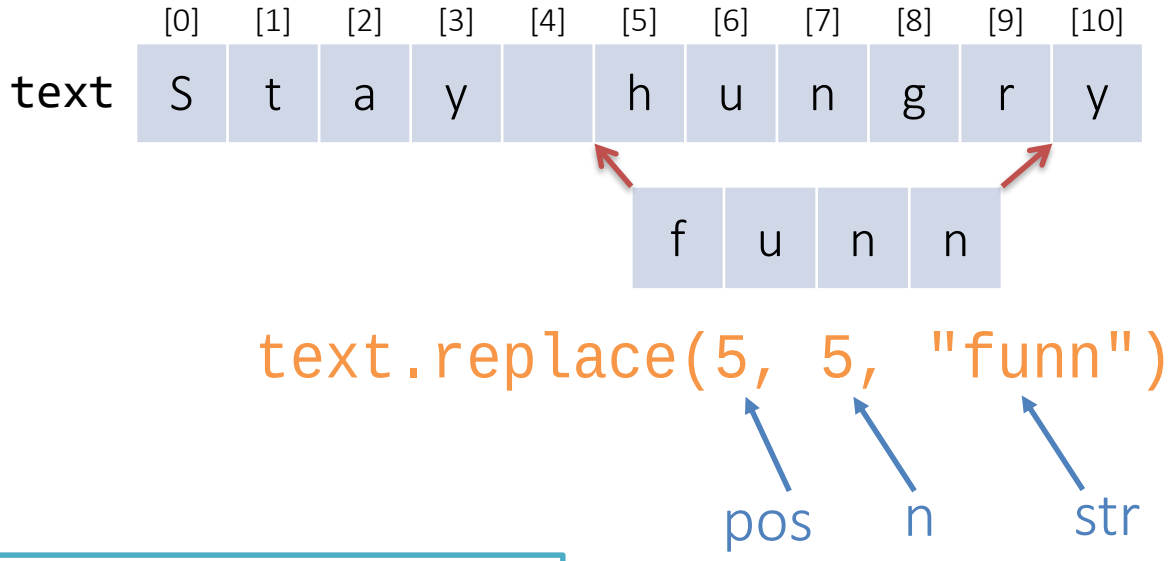
‡ Inserts the content specified in `str` at position `pos` of the current string



Resulting string: "Stay very hungry"

string::replace()

‡ Replaces `n` characters starting at position `pos` from the current string by the content specified in `str`



Resulting string: "Stay funny"

Example

```
string s1 = "Cloudy and warm.";
string s2 = "Angel is taking programming.";
string t1 = " very";
string t2 = "Nelson";

cout << s1.insert(10, t1) << endl;
cout << s2.replace(0, 5, t2) << endl;
```

string_insert_replace.cpp

```
Cloudy and very warm.
Nelson is taking programming.
```

Screen outputs

We are happy to help you!

"If you face any problems in understanding the materials,
please feel free to contact me, our TAs or student TAs.
We are very happy to help you!
We wish you enjoy learning programming in this class ☺."