Introduction

- Two string types:
  - **C-String**
    - array with base type `char`
    - end of string marked with null, `\0`
    - “older” method inherited from C
  - Class **string**
    - use templates

C-String

- Array with base type `char`
  - one character per indexed variable
  - one extra character: `\0`
    - called null character or end marker
- We’ve used C-Strings
  - literal “Hello” stored as C-String

Learning Objectives

- An Array Type for Strings
  - C-String
- Character Manipulation Tools
  - Character I/O
  - get, put member functions
  - putback, peek, ignore
  - Common `<cctype>` functions
- Standard Class string
  - string processing
C-String Variable

- Array of characters:
  - \texttt{ex: char s[10];}
  - declare a C-String variable to hold up to 9 characters
  - plus one null character ‘\0’
- Typically \textit{partially-filled} array
  - declare large enough to hold a max-size string
  - indicate end with null
- Only difference from standard array:
  - must contain null character

C-String vs. Character

- There is a difference between ‘A’ and “A”
  - ‘A’ is the character A
  - “A” is the string A
- Because strings are null terminated, “A” represents two characters, ‘A’ and ‘\0’

Examples of C-String (1/2)

- Consider the statement
  - \texttt{char name[16];}
- Because C-String are null terminated and name has sixteen components
  - The largest string that can be stored in name is 15
- If you store a string of length, say 10, in \texttt{name}
  - the first 11 components of \texttt{name} (include ‘\0’) are used and
  - the last 5 components are left unused

Examples of C-String (2/2)

- The statement
  - \texttt{char name[16] = “John”;}
  - declares a string variable \texttt{name} of length 16 and stores “John” in it
- The statement
  - \texttt{char name[] = “John”;}
  - declares a string variable \texttt{name} of length 5 and stores “John” in it
### C-String Indexes

- A C-String is an array
- Can access indexed variables of:
  ```
  char ourStr[6] = "EED";
  - ourStr[0] is 'E'
  - ourStr[1] is 'E'
  - ourStr[2] is 'D'
  - ourStr[3] is '\0'
  - ourStr[4] is unknown
  - ourStr[5] is unknown
  ```

<table>
<thead>
<tr>
<th>'E'</th>
<th>'E'</th>
<th>'D'</th>
<th>'\0'</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
</table>

### C-String Index Manipulation

- Can manipulate indexed variables
  ```
  char happyString[7] = "DoBeDo";
  ```
- Be careful!
- Now, '\0' (null) was overwritten by a 'Z'
- If null overwritten, C-String no longer acts like C-String!
- unpredictable results!
  ⇒ don’t know when to finish

### Library for C-String Manipulation

- Declaring C-String
  - Requires No C++ library
  - Built into standard C++

- Manipulations
  - Require library `<cstring`
  - Typically included when using C-String
  - Normally want to do "fun" things with them
  - Ex: copy, remove, modify and etc

### Common Functions in `<cstring>`

<table>
<thead>
<tr>
<th>type</th>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *</td>
<td><code>strcpy</code>(char *s1, const char *s2)</td>
<td>copy string</td>
</tr>
<tr>
<td>char *</td>
<td><code>strcat</code>(char *s1, const char *s2)</td>
<td>append string</td>
</tr>
<tr>
<td>int</td>
<td><code>strcmp</code>(const char *s1, const char *s2)</td>
<td>compare two strings</td>
</tr>
<tr>
<td>int</td>
<td><code>strlen</code>(const char *s)</td>
<td>return string length</td>
</tr>
<tr>
<td>char *</td>
<td><code>strchr</code>(const char *s, int c)</td>
<td>find a character in string</td>
</tr>
<tr>
<td>char *</td>
<td><code>strstr</code>(const char *s1, const char *s2)</td>
<td>find string s2 in string s1</td>
</tr>
</tbody>
</table>
C-String Assignment by `strcpy()`

- C-String not like other variables
- Cannot assign or compare:
  ```
  char mystr[10];
  mystr = "Hello";  // Error! Illegal !!!
  ⇒ can only use "=" at declarations of C-String
  ```
- Must use library function `strcpy(cstr1, cstr2)` for assignments
  ```
  Ex: strcpy(mystr, "Hello");
  ```
- NO checks for size ⇒ up to programmer, just like other arrays!

Compare C-Strings by `strcmp()`

- Also cannot use operator `==`
  ```
  char mystr_a[10] = "Hello";
  char mystr_b[10] = "Goodbye";
  -mystr_a == mystr_b;    // NOT allowed!
  ```
- Must use library function `strcmp(cstr1,cstr2)`
  ```
  Ex:
  ```
  if (strcmp(mystr_a, mystr_b))
    cout << "Strings are NOT the same.";
  else
    cout << "Strings are the same.";
  ```

Reading Results from `strcmp()`

- C-Strings are compared character by character using the collating sequence of the system
- If we are using the ASCII character set
  1. The string “Air” is smaller than the string “Boat” ⇒ ‘B’ > ‘A’
  2. The string “Air” is smaller than the string “An” ⇒ ‘n’ > ‘i’
  3. The string “Billy” is larger than the string “Bill” ⇒ ‘y’ > ‘\0’
  4. The string “Hello” is smaller than “hello” ⇒ ‘h’ > ‘H’

Compute Length by `strlen()`

- `strlen()` computes the length of the given C-String
- Example:
  ```
  char myStr[10] = "dobedo";
  cout << strlen(myStr);
  ```
- Return the number of characters but automatically excluding null character
- Result here: 6
Concatenate C-Strings by `strcat()`

- `strcat()`
  - **concatenate** two C-Strings into one C-String
  - Example:
    ```
    char mystr[20] = "The rain";
    strcat(mystr, "in Spain");
    ```
  - Note result:
    - mystr now contains "The rain in Spain"
  - Be careful!
  - Incorporate *spaces* as needed!
    ```
    strcat(mystr, " in Spain");
    ```

Search: `strchr()` and `strstr()`

- `strchr(cstr)`
  - locate *first* occurrence of character in C-String
  - Example:
    ```
    char mystr[] = "This is a simple string";
    char * pch = strchr(mystr, 's');
    // pch point to mystr[3]
    ```
- `strstr(cstr1, cstr2)`
  - return a pointer to the first occurrence of `cstr2` in `cstr1`, or a null if there `cstr2` is not in `cstr1`
  - Example:
    ```
    char * pch = strstr(mystr, "simple");
    // pch points to mystr[10]
    ```

C-String Arguments and Parameters

- Recall: a C-String is an *array* as well
  - So C-String parameters are array parameters
    - C-Strings passed to functions can be changed by receiving function!
  - Like all arrays, typical to send size as well
    - Function could also use `\0` to find end
    - So size *not necessary* if function won’t change C-String parameter
    - Use *const* modifier to protect C-String arguments

C-String Output and Input

- Can output with insertion operator, `<<` cout `<<` news `<<` " Wow.
  ```
  cout << news << " Wow.\n"
  ```
  - news is a C-String variable
  - `<<` is overloaded for C-Strings
- Can input with extraction operator, `>>`
  - Issues exist, however
    - Whitespace is a delimiter
      - tab, space, line breaks are skipped
    - Input reading stops at *any* delimiter
- Watch *size* of C-String
  - Must be large enough to hold entered string
    ```
    no warnings of such issues
    ```
C-String Input Example

- Example:
  
  ```
  char a[80], b[80];
  cout << "Enter input: ";
  cin >> a >> b;
  cout << a << b << " END OF OUTPUT\n";
  ```

- Dialogue offered:
  
  Enter input: Do be do to you!
  Do be END OF OUTPUT
  
  - Note: Underlined portion typed at keyboard
  
  - C-String `a` receives: “Do”
  
  - C-String `b` receives: “be”

Get Line Input as a C-String

- Can receive entire line into one C-String

  ```
  Use getline(), a predefined member function
  ```

  ```
  Ex:
  char mystr[80];
  cout << "Enter input: ";
  cin.getline(mystr, 80);
  cout << mystr << " END OF OUTPUT\n";
  ```

- Dialogue:
  
  Enter input: Happy Together! Happy Together! Happy Together! Happy Together! Happy Together!END OF OUTPUT

More on `getline()`

- Can explicitly tell length to receive

  ```
  char shortStr[5];
  cout << "Enter input: ";
  cin.getline(shortStr, 5);
  cout << shortStr << " END OF OUTPUT\n";
  ```

- Results:
  
  Enter input: dobedowap
dobe END OF OUTPUT

- Force FOUR characters only be read
  
  recall need for null character ‘\0’

Character I/O in C++

- Input and output data

  - ALL treated as character data
  
  - e.g., number 103 outputted as ‘1’, ‘0’ and ‘3’
  
  - Conversion done automatically ⇒ use low-level utilities

- Can use same low-level utilities ourselves as well
**Member Function `get()`**

- Read one character at a time
- Member function of `cin` object:
  
  ```cpp
char nextToken;
cin.get(nextToken);
```
- Reads next char & puts in variable `nextToken`
- Argument must be `char` type, NOT string

**Member Function `put()`**

- Output one character at a time
- Member function of `cout` object:
- Examples:
  ```cpp
cout.put('a');
```
  - Outputs letter ‘a’ to screen
  ```cpp
  char mystr[10] = "Hello";
cout.put(mystr[1]);
```
  - Outputs letter ‘e’ to screen

**More Member Functions**

- `putback()`
  - once read, might need to put back
  - Ex: `cin.putback(lastChar);`

- `peek()`
  - return next char, but leave it there
  - Ex: `peekChar = cin.peek();`

- `ignore()`
  - skip input, up to designated character
  - Ex: `cin.ignore(1000, '\n');`
  - skip at most 1000 characters until ‘\n’

**Example for `putback()`**

```cpp
char c; int n; char str[256];
cout << "Enter a number or a word: ";
c = cin.get();
if ( (c >= '0') && (c <= '9') )
{
  cin.putback(c);
  cin >> n;
  cout << "You have entered number " << n << endl;
}
else
{
  cin.putback(c);
  cin >> str;
  cout << "You have entered word " << str << endl;
}
```
Example for `peek()`

```cpp
char c; int n; char str[256];
cout << "Enter a number or a word: ";
c = cin.peek();
if ( (c >= '0') && (c <= '9') )
{
    cin >> n;
    cout << "You have entered number " << n << endl;
}
else {
    cin >> str;
    cout << "You have entered word " << str << endl;
}
```

Example for `ignore()`

```cpp
char first;
char last;
cout << "Enter first and last names: ";
first = cin.get();
cin.ignore(256, ' ');
cin.ignore(256, ' ');
cin.ignore(256, ' ');
cin.ignore(256, ' ');
last = cin.get();
cout << "Your initials are ";
cout << first;
cout << last;
```

Manipulation in `<cctype>` (1/2)

- `toupper(char)`
  - return uppercase of a character
  - Ex: `char c = toupper('a'); //c='A'
- `tolower(char)`
  - return lowercase of a character
  - Ex: `char c = tolower('B'); //c='b'
- `isupper(char)`
  - true if the character is uppercase
  - Ex: `bool cond = isupper('A'); //cond=true`
- `islower(char)`
  - true if the character is lowercase
  - Ex: `bool cond = islower('A'); //cond=false`

Manipulation in `<cctype>` (2/2)

- `isalpha(char)`
  - Check if character is alphabetic
  - Ex: `bool cond = isalpha('a'); //cond=true`
- `isdigit(char)`
  - Check if character is a digit from '0' to '9'
  - Ex: `bool cond = isdigit('M'); //cond=false`
- `isalnum(char)`
  - Check if character is either a letter or a digit
  - Ex: `bool cond = isalnum('A'); //cond=true`
- `isspace(char)`
  - Check if character is a white-space
  - Ex: `bool cond = isspace('A'); //cond=false`
Standard Class string

- Defined in library:
  ```
  #include <string>
  using namespace std;
  ```
- String variables and expressions
  - Treated much like simple types
- Can assign, compare, add:
  ```
  string s1, s2, s3;
  s3 = s1 + s2; // Concatenation
  s3 = “Hello Mom!” // Assignment
  ```
- Note C-String "Hello Mom!" automatically converted to string type!

First Example of Class string

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string sentence1;
  string adj(“beautiful”);
  string noun1(“hair”);
  string sentence2 = “Good eye!”;

  sentence1 = “You have ” + adj + “ ” + noun1;
  cout << sentence1 << endl;
  cout << sentence2 << endl;
  return 0;
}
```

Construction of Class string

- Example:
  ```
  string mystr;
  // default construction; create empty string object mystr
  string mystr(“test”);
  // create a string object with data “test”
  // equivalent to string mystr = “test”;
  string mystr(aStr);
  // create a string object and copy data of aStr
  ```

Element Access of Class string

- Example:
  ```
  mystr[i];
  // return read/write reference to character in mystr at index i
  mystr.at(i);
  // return read/write reference to character in mystr at index i
  mystr.substr(position, length);
  // return the sub-string of the calling object starting at position and having length characters
  ```
**Assignment of Class string (1/2)**

- Example:
  
  ```
  mystr1 = mystr2;
  //allocate space and initialize it to data in mystr2; release allocated memory of original mystr1; set mystr1 size to mystr2 size
  mystr1 += mystr2;
  //append mystr2 to the end of mystr1; size will be set properly
  mystr.empty();
  //return true if mystr is an empty string; otherwise, return false
  ```

**Assignment of Class string (2/2)**

- Example:
  
  ```
  mystr = mystr1 + mystr2;
  //return mystr with mystr1’s data concatenated by mystr2’s data
  mystr1.insert(position, mystr2);
  //insert mystr2 into mystr1 from the position
  mystr1.remove(position, length);
  //remove a substring of length from position in mystr1
  ```

**Comparison of Class string**

- Example:
  
  ```
  (mystr1 == mystr1) or (mystr1 != mystr1)
  //test for equality or inequality, return 0 or 1
  (mystr1 > mystr1) or (mystr1 < mystr1)
  (mystr1 >= mystr1) or (mystr1 <= mystr1)
  //test for lexicographical comparison, return 0 or 1
  mystr1.find(mystr2) or
  mystr1.find(mystr2, position)
  //return the index of first occurrence of mystr2 in mystr1 (from the position)
  ```

**I/O with Class string**

- Just like other types
  
  ```
  string s1, s2;
  cin >> s1;
  cin >> s2;
  //User types in:
  Today is a happy day!
  ```
  
  ```
  //Extraction still ignores whitespace
  -s1 receives value “Today”
  -s2 receives value “is”
  ```
getline() with Class string

- For getting complete lines:

```cpp
string line;
cout << "Enter a line of input: ";
getline(cin, line);
cout << line << "END OF OUTPUT";
```

- Dialogue produced:
  Enter a line of input: **Do be do to you!**
  Do be do to you!END OF INPUT
  - similar to C-String's usage of getline()

Other getline() Versions

- Can specify *delimiter* character:

```cpp
string line;
cout << "Enter input: ";
groupline(cin, line, "?"); 
  - Receives input until "?" encountered
```

- getline() actually returns *reference*
  - Ex:
    ```cpp
    string s1, s2;
groupline(cin, s1) >> s2;
    ```
  - Results in:
    1. First, s1 gets a line and
    2. Next, cin >> s2; // s2 gets a string

Pitfall: Mixing Input Methods

- Be careful mixing `cin >> var` and `getline()`

  - Example:
    ```cpp
    int n;
    string strline;
    cin >> n;
groupline(cin, strline);
    ```
  - If the input is:
    42
    Hello! hitchhiker.
    - variable n set to 42
    - strline set to empty string!
  - `cin >> n` skipped leading whitespace, leaving '\n' on stream for getline()

Class string Processing

- Same operations available as C-Strings
- And more!
  - Over 100 members of standard string class
- Some member functions:
  - length()
    - return length of string variable
  - at(i)
    - return reference to char at position i
- Example:
  ```cpp
  string str = "myClass";
cout << str.length() << " " << str.at(2); ```
C-String and string Conversions

- Automatic type conversions
  - From C-String to string object:
    ```
    char aCString[] = "My C-string";
    string stringVar;
    stringVar = aCstring;
    ```
  - perfectly legal and appropriate!
  - ```
    aCString = stringVar;
    ```
  - illegal! Cannot auto-convert to C-String

- Must use explicit conversion:
  ```
  strcpy(aCString, stringVar.c_str());
  ```

Summary

- C-String variable is "array of characters"
  - With addition of null character, '\0'
- C-Strings act like arrays
  - Cannot assign, compare like simple variables
- Libraries `<cctype>` & `<string>` have useful manipulating functions
- `cin.get()` reads next single character
- `getline()` versions allow full line reading
- Class string objects are better-behaved than C-Strings