UCLA

CS 31:
Introduction To Computer Science I

Howard A. Stahl

SANTA MONICA COLLEGE

Agenda

- CStrings
- `cstring` Library Functions
- `cstdlib` Library Functions
- `cctype` Library Functions
- CStrings As Function Arguments
- `string` Class

Text Data

- C++ Provides Two Core String Datatypes
- C++ Provides "C-Strings"
  - String Datatype From The C Language Point-Of-View
  - A Null-Terminated Array of `char`
- C++ Provides The `string` Class To Manipulate Text Data
  - You Must `#include <string>`

CStrings

- CStrings Are Arrays Of Characters
- CStrings Are How C Programs Manipulate Text
- We've Already Been Using CStrings
  - The Literal "Hello World" Is Stored As A CString

CStrings

- CStrings Are, By Convention, Always Null Terminated
  - NULL is a special symbol: '\0'
- You Must Have Enough Space In Your Array For This Extra Character!
- CString Arrays Are Partially Filled Arrays
- CStrings Are Arrays With An Extra Letter

CStrings

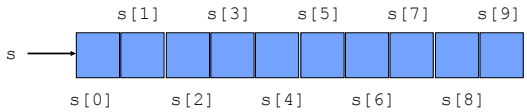
- Example:

```
char s[ 10 ];
```

CStrings

- Example:

```
char s[ 10 ];
```



CStrings

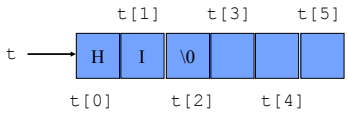
- Example:

```
char t[ 5 ] = "HI";
```

CStrings

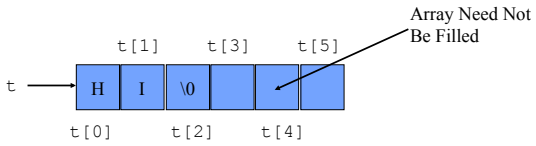
- Example:

```
char t[ 5 ] = "HI";
```



CStrings

- Example:
  `char t[ 5 ] = "HI";`

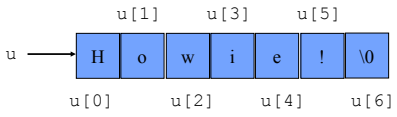


CStrings

- Example:
  `char u[] = "Howie!";`

CStrings

- Example:
  `char u[] = "Howie!";`

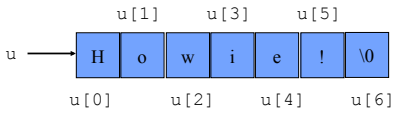


CStrings

- Example:

```
char u[] = "Howie!";
```

Size Is Not Required



CStrings Observations

- CStrings Are Implemented As Arrays Of Char
- Unfortunately, CStrings Are Used In A Different Way And Do Not Support Your Intuition, As You Will See
  - the sentinel \0 is vital to cstrings working right!
- CStrings Are Tricky!

CStrings CounterIntuition #1

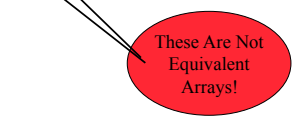
- Example:

```
char v[] = "abc";
char w[] = {'a', 'b', 'c'};
```

CStrings CounterIntuition #1

- Example:

```
char v[] = "abc";
char w[] = {'a', 'b', 'c'};
```



CString CounterIntuition #2

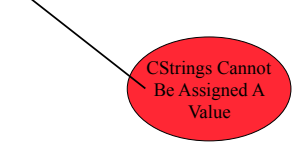
- Example:

```
char x[ 5 ];
x = "Foo";
```

CString CounterIntuition #2

- Example:

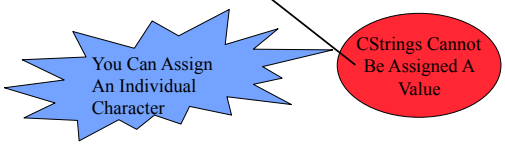
```
char x[ 5 ];
x = "Foo";
```



CString CounterIntuition #2

- Example:

```
char x[ 5 ];
x = "Foo";
```



CString Is An Array

- You Can Use An Index To Walk The Array
- Example:

```
char lots_of_x[] = "Hello World";

int index = 0;
while (lots_of_x[ index ] != '\0') {
   lots_of_x[ index ] = 'x';
   index++;
}
```

CString Is An Array

- Be Very Careful Never To Write Over The Ending NULL Character
  - If You Do, You MUST Put A NULL Character Onto The End Of Your Array
  - Otherwise, You Will Get Unpredictable Results

Time For Our First Demo!

- CString.cpp

(See Handout For Example 1)

Summarizing Our First Demo!

- Always Remember About \0 When Working With CStrings
- You Cannot Use Assignment With CStrings
- CStrings Can Be Declared `const` If Appropriate

<cstring> Library

Display 9.1 **Some Predefined C-String Functions in <cstring>**

FUNCTION	DESCRIPTION	CAUTIONS
`strcpy(`*Target_String_Var*`,` *Src_String*`)`	Copies the C-string value *Src_String* into the C-string variable *Target_String_Var*.	Does not check to make sure *Target_String_Var* is large enough to hold the value *Src_String*.
`strcpy(`*Target_String_Var*`,` *Src_String*`,` *Limit*`)`	The same as the two-argument `strcpy` except that at most *Limit* characters are copied.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcpy`. Not implemented in all versions of C++.
`strcat(`*Target_String_Var*`,` *Src_String*`)`	Concatenates the C-string value *Src_String* onto the end of the C-string in the C-string variable *Target_String_Var*.	Does not check to see that *Target_String_Var* is large enough to hold the result of the concatenation.

(continued)

<cstring> Library

Display 9.1 **Some Predefined C-String Functions in <cstring>**

FUNCTION	DESCRIPTION	CAUTIONS
`strcat(`*Target_String_Var*`,` *Src_String*`,` *Limit*`)`	The same as the two argument `strcat` except that at most *Limit* characters are appended.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcat`. Not implemented in all versions of C++.
`strlen(`*Src_String*`)`	Returns an integer equal to the length of *Src_String*. (The null character, '\0', is not counted in the length.)	
`strcmp(`*String_1*`,` *String_2*`)`	Returns 0 if *String_1* and *String_2* are the same. Returns a value < 0 if *String_1* is less than *String_2*. Returns a value > 0 if *String_1* is greater than *String_2* (that is, returns a nonzero value if *String_1* and *String_2* are different). The order is lexicographic.	If *String_1* equals *String_2*, this function returns 0, which converts to `false`. Note that this is the reverse of what you might expect it to return when the strings are equal.
`strcmp(`*String_1*`,` *String_2*`,` *Limit*`)`	The same as the two-argument `strcat` except that at most *Limit* characters are compared.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcmp`. Not implemented in all versions of C++.

`cstring` System Library

- The `cstring` System Library Provides Access To Various C Library String Manipulation Routines

FUNCTION	MEANING	ARGUMENTS	RETURNS
strcpy(dest, src)	dest = src	cstring, cstring	void
strcat(dest, src)	dest = dest + src	cstring, cstring	void
strlen(src)	length of cstring src	cstring	int
strcmp(s1, s2)	compares s1 and s2	cstring, cstring	int

See Appendix 4 For A Fuller List

`cstring` System Library

- `strlen( char s[] ) :: int`
  - returns length of s NOT including \0
- `strcmp(char s[],char t[])::int`
  - returns 0 if s equals t lexicographically
  - returns <0 if s is lexicographically less than t
  - returns >0 if s is lexicographically greater than t
  - NOTE: `0` equates to `false` in expressions

`cstdlib` System Library

- The `cstdlib` System Library Provides Access To Various Conversion Routines

FUNCTION	MEANING	ARGUMENTS	RETURNS
atoi(src)	ascii to integer	cstring	int
atof(src)	ascii to floating-point	cstring	double
atol(src)	ascii to long	cstring	long

See Appendix 4 For A Fuller List

CString Functions: strlen()

- "String Length"
- Often Useful To Know A String's Length

```
char myString[10] = "dobedo";
cout << strlen(myString);
```

- Returns Number Of Characters Not Including NULL
- Result here: `6`

CString Functions: strcat()

- "String concatenate"
- Essentially += On Two Pieces Of Text

```
char stringVar[20] = "The rain";
strcat(stringVar, "in Spain");
```

- Result: stringVar = "The rainin Spain"
- Be Careful! Have Enough Space For The Result And An Ending NULL!
- Add spaces as needed!

<cctype> System Library

- Another Useful Library Of Functions

Function	Argument	Result
toupper(c)	char	int
tolower(c)	char	int
isupper(c)	char	bool
islower(c)	char	bool
isalpha(c)	char	bool
isdigit(c)	char	bool
isspace(c)	char	bool

<cctype> System Library

- Another Useful Library Of Functions

Function	Argument	Result
toupper(c)	char	int
tolower(c)	char	int
isupper(c)	char	bool
islower(c)	char	bool
isalpha(c)	char	bool
isdigit(c)	char	bool
isspace(c)	char	bool

<cctype> System Library

- Example:
  ```
  char c = toupper( 'a' );
  cout << c;
  ```
  Although function returns `int`, it can be cast back into a `char`, as we expect
- Alternatively,
  ```
  cout << char( toupper('a') );
  ```

CStrings As Function Parameters

- Under The Covers, A CString Is An Array
- CStrings Passed To A Function Can Be Changed By The Function And The Caller Will See The Result
- If It Is A Read-Only Argument, Mark The CString With `const`
- Size Companion Parameter Only Needed If The String Will Get Changed

CString Input

- We Have Two Choices:
  - Use >> With `cin` Which Eats Leading Whitespace And Reads Just A Single Word
  - Use `cin.getline` Which Is The Version That Supports CString Data

CString Input

- `cin.getline( char *, int )`

```
char a[80];
cout << "Enter input: ";
cin.getline(a, 80);
cout << a << endl;
```

- Result: Enter input: Do be do to you!
  Do be do to you!

CStrings Are Nasty!

- Example:

```
char var[3];
strcpy( var, "A Nice Long String" );
```

CStrings Are Nasty!

- Example:

```
char var[3];
strcpy( var, "A Nice Long String" );
```

You Must Use strcpy To Do Assignments

CStrings Are Nasty!

- Example:

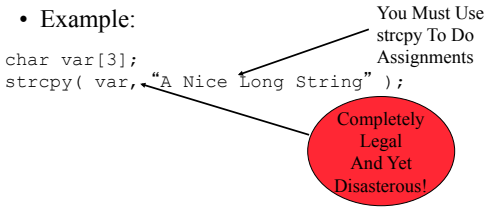
```
char var[3];
strcpy( var, "A Nice Long String" );
```

You Must Use strcpy To Do Assignments

Completely Legal And Yet Disasterous!

CStrings Are Nasty!

- Example:



It Is Always The Programmer's Responsibility To Ensure That The Destination Is Large Enough For The String Being Placed There

CStrings As Function Arguments

- Like Other Array Variables, CStrings Can Be Passed To Functions
- Generally, It's A Good Idea To Provide An Argument That Specifies The Maximum String Size Allowed
- Read-Only Functions Can Use The \0 Sentinel To Determine The End Of The String

Understanding CString Parameters

- Read-Only Functions Can Work Off The \0 Sentinel Embedded In A CString
- Functions Updating A CString Variable Should Be Provided A Maximum Size Allowed Value As A Parameter
- Use `strcmp` Carefully!
  - return `0` when the two cstrings are equal
  - `0` equates to `false`

Stream Input

- By Default, Stream Insertion Operator Eats Whitespace
- But Whitespace Is Meaningful To Strings
- To Read Character Data, Use `getline` Function

Stream Input

- Example: `istream::getline(char s[],int i)` reads up to i-1 chars into s, stops at new-line

```
const int LINESIZE=80;
char line1[LINESIZE];
char line2[LINESIZE];

cin.getline( line1, LINESIZE );
cin.getline( line2, LINESIZE );
```

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace

```
int number;
char str[80];

cin >> number;
cin.getline( str, 80 );
```

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace

```
int number;
char str[80];

cin >> number;
cin.getline( str, 80 );
```

INPUT
30
data

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace

```
int number;
char str[80];

cin >> number;
cin.getline( str, 80 );
```

INPUT
30
data

number

str

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace

```
int number;
char str[80];

cin >> number;
cin.getline( str, 80 );
```

INPUT
30
data

number
30

str
""

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace
- Use `istream::ignore( int, char )`

```
int number;
char str[80];

cin >> number;
cin.ignore( 10000, '\n' );
cin.getline( str, 80 );
```

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace
- Use `istream::ignore( int, char )`

```
int number;
char str[80];

cin >> number;
cin.ignore( 10000, '\n' );
cin.getline( str, 80 );
```

INPUT
30
data

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace
- Use `istream::ignore( int, char )`

```
int number;
char str[80];

cin >> number;
cin.ignore( 10000, '\n' );
cin.getline( str, 80 );
```

INPUT
30
data

number

str

Stream Input

- Unfortunately, Things Are More Involved...
- When Intermixing >> With `getline`, Note That `cin` Eats Only Opening Whitespace
- Use `istream::ignore( int, char )`

```
int number;
char str[80];

cin >> number;
cin.ignore( 10000, '\n' );
cin.getline( str, 80 );
```

INPUT
30
data

number
30

str
"data"

`string` Class

- A Standard Class From The `string` Library
- + Operator Performs Concatenation
- Default And String Argument Constructor
- Allows For Character Access Using `[]` Indexing
- << And >> Are Overloaded As You Expect
- All Boolean Operators Work As You Expect

`string` Class

- Example:

```
#include <string>

string name, dog("dog"), hotdog;
cin  >> name;
hotdog = "hot " + dog;

for (int i=0; i < name.length(); ++i)
    cout << name[i] << " ";
```

`string` Class

- Example:

```
#include <string>

string name, dog("dog"), hotdog;
cin  >> name;
hotdog = "hot " + dog;

for (int i=0; i < name.length(); ++i)
    cout << name[i] << " ";
```

[ ] operator does not perform bounds checking

`string` Class

- Example:

```
#include <string>

string name, dog("dog"), hotdog;
cin  >> name;
hotdog = "hot " + dog;

for (int i=0; i < name.length(); ++i)
    cout << name[i] << " ";
```

[ ] operator does not perform bounds checking

consider using .at( ) member

`getline` Function For `string` Objects

- `getline()` Function For `string` Objects Is A Normal Function
  - not a member of istream

```
string& getline( istream& input,
                 string& str,
                 char delimiter = '\n' );
```

`getline` Function For `string` Objects

- `getline()` Function For `string` Objects Is A Normal Function
  - not a member of istream

same icky issues with mixing >> and getline exist

```
string& getline( istream& input,
                 string& str,
                 char delimiter = '\n' );
```

`string` Member Functions

- `string` Class Is Pretty Functional!

FUNCTION	MEANING	ARGUMENTS	RETURNS
substring(pos, len)	substring starting at pos for length len	int, int	string
empty( )	tests for empty		boolean
insert(pos, str)	insert string at pos	int, string	void
remove(pos, len)	remove starting at pos for length len	int, int	void
find(str)	find first occurrence of str in instance	string	int

See Page 650 For A Fuller List

Time For Our Next Demo!

- String.cpp

(See Handout For Example 2)

Summarizing Our Second Demo!

- `string` Class Is Much Nicer To Deal With!
- `string` Class Follows Our Intuition
- Remember That Boolean Operators Work Different With CString And `string` Objects!

UCLA

Summary

- CStrings
- `cstring` Library Functions
- `cstdlib` Library Functions
- `cctype` Library Functions
- CStrings As Function Arguments
- `string` Class