
Advance C++
Lecture 3

The switch Statement

- A statement for controlling multiple branches
- Can do the same thing with if statements but sometimes switch is more convenient
- Uses controlling expression which returns bool data type (true or false)
- Syntax:
  - Next slide

switch Statement Syntax

switch Statement

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
                    .
                    .
                    .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}
```

You need not place a `break` *statement in each case. If you omit a* `break`*, that case continues until a* `break` *(or the end of the* `switch` *statement) is reached.*

The controlling expression must be integral! This includes char.

The switch Statement in Action

EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
}
```

If you forget this **break**, *then passenger cars will pay $1.50.*

The switch: multiple case labels

- Execution "falls thru" until break
  - switch provides a "point of entry“
- **Example:**

```
case 'A':
case 'a':
    cout << "Excellent: you got an "A"!\n";
    break;
case 'B':
case 'b':
    cout << "Good: you got a "B"!\n";
    break;
```

  - Note multiple labels provide same "entry"

switch Pitfalls/Tip

- Forgetting the break;
  - No compiler error
  - Execution simply "falls thru" other cases until break;
- Biggest use: MENUs
  - Provides clearer "big-picture" view
  - Shows menu structure effectively
  - Each branch is one menu choice

switch Menu Example

- Switch stmt "perfect" for menus:

```
switch (response)
{
 case 1:
      // Execute menu option 1
      break;
 case 2:
      // Execute menu option 2
      break;
 case 3:
      // Execute menu option 3
      break;
 default:
      cout << "Please enter valid response.";
}
```


Strings

Learning Objectives

- An Array Type for Strings
  - C-Strings
- Character Manipulation Tools
  - Character I/O
  - get, put member functions
- Standard Class string
  - String processing

Introduction

- Two string types:
- C-strings
  - Array with base type char
  - End of string marked with null, "\0"
  - "Older" method inherited from C
- String class
  - Uses templates

C-Strings

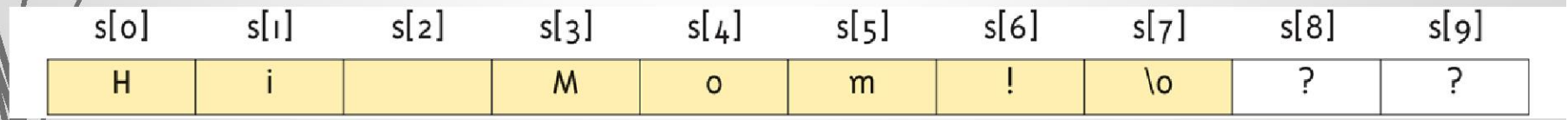
- Array with base type *char*
  - One character per indexed variable
  - One extra character: "\0"
    - Called "null character"
    - End marker
- We've used c-strings
  - Literal "Hello" stored as c-string

C-String Variable

- Array of characters:
  char s[10];
  - Declares a c-string variable to hold up to 9 characters
  - + one null character
  - Declare large enough to hold max-size string
  - Indicate end with null
- Only difference from standard array:
  - Must contain null character

C-String Storage

- A standard array:
  char s[10];
  - If s contains string "Hi Mom!", stored as:



C-String Initialization

- Can initialize c-string:
  char myMessage[20] = "Hi there.";
  - Needn't fill entire array
  - Initialization places "\0" at end
- Can omit array-size:
  char shortString[] = "abc";
  - Automatically makes size one more than length of quoted string
  - NOT same as:
    char shortString[] = {"a", "b", "c"};

C-String Indexes

- A c-string IS an array
- Can access indexed variables of:
  char ourString[5] = "Hi";
  - ourString[0] is "H"
  - ourString[1] is "i"
  - ourString[2] is "\0"
  - ourString[3] is unknown
  - ourString[4] is unknown

C-String Index Manipulation

- Can manipulate indexed variables
  char happyString[7] = "DoBeDo";
  happyString[6] = "Z";
  - Be careful!
  - Here, "\0" (null) was overwritten by a "Z"!
- If null overwritten, c-string no longer "acts" like c-string!
  - Unpredictable results!

Library

- Declaring c-strings
  - Requires no C++ library
  - Built into standard C++
- Manipulations
  - Require library <cstring>
  - Typically included when using c-strings

= and == with C-strings

- C-strings not like other variables
  - Cannot assign or compare:
    char aString[10];
    aString = "Hello"; **// ILLEGAL!**
    - Can ONLY use "=" at declaration of c-string!
- Must use library function for assignment:
  strcpy(aString, "Hello");
  - Built-in function (in <cstring>)
  - Sets value of aString equal to "Hello"
  - NO checks for size!
    - Up to programmer, just like other arrays!

Comparing C-strings

- Also cannot use operator ==

```
char aString[10] = "Hello";
char anotherString[10] = "Goodbye";
```

  - **aString == anotherString; // NOT allowed!**

- Must use **library function** again:

```
if (strcmp(aString, anotherString))
 cout << "Strings NOT same.";
else
 cout << "Strings are same.";
```

The <cstring> Library: Some Predefined C-String Functions

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)

The <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++.

C-string Functions: strlen()

- "String length"
- Often useful to know string length:
  char myString[10] = "dobedo";
  cout << strlen(myString);
  - Returns number of characters
    - Not including null
  - Result here:

6

C-string Functions: strcat()

- strcat()
- "String concatenate":
  char stringVar[20] = "The rain";
  strcat(stringVar, "in Spain");
  - Note result:
    stringVar now contains "The rainin Spain"
  - Be careful!
  - Incorporate spaces as needed!

C-string Arguments and Parameters

- Recall: c-string is array
- So c-string parameter is array parameter
  - 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

C-String Output

- Can output with insertion operator, <<
- As we've been doing already:
  cout << news << " Wow.\n";
  - Where *news* is a c-string variable
- Possible because << operator is overloaded for c-strings!

C-String Input

- Can input with extraction operator, >>
- Whitespace is "delimiter"
  - Input reading "stops" at delimiter
- Watch size of c-string
  - Must be large enough to hold entered string!
  - C++ gives no warnings of such issues!

C-String Input 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!
  DobeEND OF OUTPUT
  - Note: Underlined portion typed at keyboard
- C-string *a* receives: "do"
- C-string *b* receives: "be"

C-String Line Input

- Can receive entire line into c-string
- Use getline(), a predefined member function:

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

  - Dialogue:
    Enter input: Do be do to you!
    Do be do to you!END OF INPUT

Example: Command Line Arguments

- Programs invoked from the command line (e.g. a UNIX shell, DOS command prompt) can be sent arguments
  - Example: COPY C:\FOO.TXT D:\FOO2.TXT
    - This runs the program named "COPY" and sends in two C-String parameters, "C:\FOO.TXT" and "D:\FOO2.TXT"
    - It is up to the COPY program to process the inputs presented to it; i.e. actually copy the files
- Arguments are passed as an array of C-Strings to the main function

Example: Command Line Arguments

- Header for main
  - int main(int argc, char *argv[])
  - **argc** specifies how many arguments are supplied. The name of the program counts, so argc will be at least 1.
  - **argv** is an array of C-Strings.
    - argv[0] holds the name of the program that is invoked
    - argv[1] holds the name of the first parameter
    - argv[2] holds the name of the second parameter
    - Etc.

Example: Command Line Arguments

```
// Echo back the input arguments
int main(int argc, char *argv[])
{
  for (int i=0; i<argc; i++)
  {
    cout << "Argument " << i << " " << argv[i] << endl;
  }
  return 0;
}
```

Sample Execution

```
> Test
Argument 0 Test
```

Invoking Test from command prompt

Sample Execution

```
> Test hello world
Argument 0 Test
Argument 1 hello
Argument 2 world
```

More getline()

- Can explicitly tell length to receive:

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

- Results:
  Enter input: dobedowap
  dobeEND OF OUTPUT
- Forces FOUR characters only be read
  - Recall need for null character!

Character I/O

- Input and output data
  - ALL treated as character data
  - e.g., number 10 outputted as "1" and "0"
  - Conversion done automatically
    - Uses low-level utilities
- Can use same low-level utilities ourselves as well

Member Function get()

- Reads one char at a time
- Member function of cin object:
  char nextSymbol;
  cin.get(nextSymbol);
  - Reads next char & puts in variable nextSymbol
  - Argument must be char type
    - Not "string"!

Member Function put()

- Outputs one character at a time
- Member function of cout object:
- Examples:

```
cout.put("a");
```

  - Outputs letter "a" to screen

```
char myString[10] = "Hello";
cout.put(myString[1]);
```

  - Outputs letter "e" to screen

Character-Manipulating Functions: Some Functions in <cctype>

Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE
`toupper(`*Char_Exp*`)`	Returns the uppercase version of *Char_Exp* (as a value of type `int`).	`char c = toupper('a');` `cout << c;` **Outputs:** A
`tolower(`*Char_Exp*`)`	Returns the lowercase version of *Char_Exp* (as a value of type `int`).	`char c = tolower('A');` `cout << c;` **Outputs:** *a*
`isupper(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is an uppercase letter; otherwise, returns `false`.	`if (isupper(c))` `  cout << "Is uppercase.";` `else` `  cout << "Is not uppercase.";`

Character-Manipulating Functions: Some Functions in <cctype>

Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE
`islower(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a lowercase letter; otherwise, returns `false`.	`char c = 'a';` `if (islower(c))` `  cout << c << " is lowercase.";` **Outputs:** `a is lowercase.`
`isalpha(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a letter of the alphabet; otherwise, returns `false`.	`char c = '$';` `if (isalpha(c))` `  cout << "Is a letter.";` `else` `  cout << "Is not a letter.";` **Outputs:** `Is not a letter.`
`isdigit(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is one of the digits `'0'` through `'9'`; otherwise, returns `false`.	`if (isdigit('3'))` `  cout << "It's a digit.";` `else` `  cout << "It's not a digit.";` **Outputs:** `It's a digit.`
`isalnum(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is either a letter or a digit; otherwise, returns `false`.	`if (isalnum('3') && isalnum('a'))` `  cout << "Both alphanumeric.";` `else` `  cout << "One or more are not.";` **Outputs:** `Both alphanumeric.`

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!

Program Using the Class string

Display 9.4 Program Using the Class string

```
1  //Demonstrates the standard class string.
2  #include <iostream>
3  #include <string>
4  using namespace std;

5  int main( )
6  {
7       string phrase;
8       string adjective("fried"), noun("ants");
9       string wish = "Bon appetite!";

10      phrase = "I love " + adjective + " " + noun + "!";
11      cout << phrase << endl
12           << wish << endl;

13      return 0;
14 }
```

Initialized to the empty string.

Two equivalent ways of initializing a string variable

SAMPLE DIALOGUE

```
I love fried ants!
Bon appetite!
```

I/O with Class string

- Just like other types!
- string s1, s2;
  cin >> s1;
  cin >> s2;
- Results:
  User types in:
  GOD bless you.
- Extraction still ignores whitespace:
  s1 receives value “GOD"
  s2 receives value “bless"

getline() with Class string

- For complete lines:

```
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:
  string line;
  cout << "Enter input: ";
  getline(cin, line, "?");
  - Receives input until **"?"** encountered
- getline() actually returns reference
  - string s1, s2;
    getline(cin, s1) >> s2;
  - Results in: (cin) >> s2;

Class string Processing

- Same operations available as c-strings
- And more!
  - Over 100 members of standard string class
- Some member functions:
  - .length()
    - Returns length of string variable
  - .at(i)
    - Returns reference to char at position i

Member Functions of the Standard Class string

Display 9.7 Member Functions of the Standard Class `string`

EXAMPLE	REMARKS
Constructors	
`string str;`	Default constructor; creates empty `string` object `str`.
`string str("string");`	Creates a `string` object with data `"string"`.
`string str(aString);`	Creates a `string` object `str` that is a copy of `aString`. `aString` is an object of the class `string`.
Element access	
`str[i]`	Returns read/write reference to character in `str` at index `i`.
`str.at(i)`	Returns read/write reference to character in `str` at index `i`.
`str.substr(position, length)`	Returns the substring of the calling object starting at `position` and having `length` characters.
Assignment/Modifiers	
`str1 = str2;`	Allocates space and initializes it to `str2`'s data, releases memory allocated for `str1`, and sets `str1`'s size to that of `str2`.
`str1 += str2;`	Character data of `str2` is concatenated to the end of `str1`; the size is set appropriately.
`str.empty( )`	Returns `true` if `str` is an empty `string`; returns `false` otherwise.

(continued)

Member Functions of the Standard Class string

Display 9.7 Member Functions of the Standard Class `string`

EXAMPLE	REMARKS
`str1 + str2`	Returns a `string` that has `str2`'s data concatenated to the end of `str1`'s data. The size is set appropriately.
`str.insert(pos, str2)`	Inserts `str2` into `str` beginning at position `pos`.
`str.remove(pos, length)`	Removes substring of size `length`, starting at position `pos`.
Comparisons	
`str1 == str2  str1 != str2`	Compare for equality or inequality; returns a Boolean value.
`str1 < str2  str1 > str2`	Four comparisons. All are lexicographical comparisons.
`str1 <= str2  str1 >= str2`	
`str.find(str1)`	Returns index of the first occurrence of `str1` in `str`.
`str.find(str1, pos)`	Returns index of the first occurrence of string `str1` in `str`; the search starts at position `pos`.
`str.find_first_of(str1, pos)`	Returns the index of the first instance in `str` of any character in `str1`, starting the search at position `pos`.
`str.find_first_not_of (str1, pos)`	Returns the index of the first instance in `str` of any character *not* in `str1`, starting search at position `pos`.

C-string and string Object 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