cis15
advanced programming techniques, using c++
fall 2007
lecture # III.1

topics:

- C-style strings
- input and output

resources:

- Pohl, chapter 9

C-style strings (1).

- storing multiple characters in a single variable
- data type is still `char`
- BUT it has a *length*
- last character the is *terminator*: `'\0'`, aka `NULL`
- string constants are surrounded by *double* quotes: `"`
- example:

```
char s[6] = "ABCDE";
```

C-style strings (2).

- example:

```
char s[6] = "ABCDE";
```

- storage looks like this:

A	B	C	D	E	\0

- so with strings, you really only access the values stored at indeces 0 through $length - 1$, (or size - 2) since the value stored at $length$ is always `\0`

C-style strings (3).

- printing strings
- format sequence: `%s`
- example:

```
#include <stdio.h>
int main() {
  char str[6] = "ABCDE";
  printf( "str = %s\n", str );
} /* end of main() */
```

- output:

```
ABCDE
```

C string library (1).

- to use the string library, include the header in your C source file:

  `#include <string.h>`
- string length function:

  `int strlen( char *s );`

  this function returns the number of characters in `s`; note that this is NOT the same thing as the number of characters allocated for the string array
- string comparison function:

  `int strcmp( const char *s1, const char *s2 );`

  "This function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared."
- for more information and more string functions, do (e.g.):

  `unix$ man strcmp`

C string library (2).

- copying functions:

  `char *strcpy( char *dest, char *source );`
  - copies characters from `source` array into `dest` array up to NULL

  `char *strncpy( char *dest, char *source, int num );`
  - copies characters from `source` array into `dest` array; stops after `num` characters (if no NULL before that); appends NULL

C string library (3).

- search functions:

  `char *strchr( const char *source, const char ch );`
  - returns pointer to first occurrence of `ch` in `source`; NULL if none

  `char *strstr( const char *source, const char *search );`
  - return pointer to first occurrence of `search` in `source`

C string library (4).

- parsing function:

  `char *strtok( char *s1, const char *s2 );`
  - breaks string `s1` into a series of *tokens*, delimited by `s2`
  - called the first time with `s1` equal to the string you want to break up
  - called subsequent times with NULL as the first argument
  - each time is called, it returns the next token on the string
  - returns null when no more tokens remain

```
char inputline[1024];
char *name, *rank, *serial_num;
printf( "enter name+rank+serial number: " );
scanf( "%s", inputline );
name = strtok( inputline,"+" );
rank = strtok( null,"+" );
serial_num = strtok( null,"+" );
```

input and output.

- input and output is typically abbreviated "I/O"
- standard C I/O is handled in `stdio.h`
  which can be implemented in C++ as is or via `cstdio`
- *stream* I/O — like a "stream" of bytes flowing in or out of the computer — is handled in C++ using `iostream`

output class: ostream.

- the standard output stream is *ostream*
- it's declared in the `iostream` header
- prototypes for public members:

```
ostream& operator<<(int i);
ostream& operator<<(long i);
ostream& operator<<(double x);
ostream& operator<<(char c);
ostream& operator<<(const char *s);
ostream& put(char c);
ostream& write(const char *p, int n);
ostream& flush();
```

- you have already been using `cout`, as in:

```
cout << "hello world!\n";
```

- new functions are:
  - `put()`, which outputs a single character
  - `write()`, which outputs N characters of a string
  - `flush()`, which forces any pending characters for the stream to be output

formatted output.

- you can use newline (`\n` or `endl`) and tab (`\t`) to format output nicely, as well as space
- be aware of *fixed width* versus *variable width* fonts when planning formatted output...
- there are some formatting functions in the `ostream` class:
  `setf()`, `precision()`, `width()`
- C++ also has a set of "manipulator" functions in `iomanip`
- some public functions:
  - `scientific`, which prints numbers using scientific notation
  - `left`, which left justifies output
  - `right`, which right justifies output
  - `setw( int )`, which sets the width of the output field
  - `setfill( int )`, which sets the "fill" character
  - `setbase( int )`, which sets the base format
  - `setprecision( int )`, which sets floating point precision

formatted output: example 1.

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

int main() {
  const int A = 5;
  const double B = 3.4568;
  double C;
  cout << "output using fixed precision, 2 decimal places:\n";
  cout.setf( ios::fixed, ios::floatfield );
  cout.precision( 2 );
  cout << "B=" << B << endl;
  cout << "output using width=10, left justified:\n";
  cout.setf( ios::left );
  cout.width( 10 );
  cout << "B=" << B << endl;
  cout << "output using width=10, right justified:\n";
```

```
  cout.setf( ios::right );
  cout.width( 10 );
  cout << "B=" << B << endl;
  cout << "you have to repeat the formatting if you want the same thing again:\
  C = sin( B );
  cout.setf( ios::right );
  cout.width( 10 );
  cout << "C=" << C << endl;
} // end of main()

SAMPLE OUPUT:

output using fixed precision, 2 decimal places:
B=3.46
output using width=10, left justified:
B=        3.46
output using width=10, right justified:
        B=3.46
you have to repeat the formatting if you want the same thing again:
        C=-0.31
```

formatted output: example 2.

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

int main() {
  long double r;
  cout << "Enter radius: ";
  cin >> r;
  cout << "no formatting:          area=" << r*r << endl;
  cout << "width:                  area=" << setw(20) << r*r << endl;
  cout << "width and precision:    area="
       << setw(20) << setprecision(10) << r*r << endl;
  cout << "width, precision, fill: area=" << setfill('*')
       << setw(20) << setprecision(10) << r*r << endl;
} // end of main()
```

```
SAMPLE OUTPUT:

Enter radius: 34
no formatting:          area=1156
width:                  area=                1156
width and precision:    area=                1156
width, precision, fill: area=****************1156
```

output and user-defined types

- typically, user-defined types (i.e., classes) have a `print()` function that specify how the data members of that type should be output
- example:

```
class point {
public:
  void print() const {
    cout << "(" << x << "," << y << ")" << endl;
  } // end of print()
private:
  int x, y;
} // end of class point
```

input class: istream.

- the standard input stream is *istream*
- it's declared in the `iostream` header
- prototypes for public members:

```
istream& operator>>(int& i);
istream& operator>>(long& i);
istream& operator>>(double& x);
istream& operator>>(char& c);
istream& operator>>(char *s);
istream& get( char &c );
istream& get( char *s, int n, char c='\n' );
istream& getline( char *s, int n, char c='\n' );
istream& read( char *s, int n );
```

- you have already been using `cin`, as in:

```
int i;
cout << "enter a number: ";
cin >> i;
```

- new functions are:
  - `get()`, which reads in either a single character or a string of specified length
  - `getline()`, which reads in a line (string) of specified length
  - `read()`, which also reads in a string of specified length
- the functions that have `n` as a parameter, read in `n-1` characters from the keyboard and put a NULL (`\0`) string termination character in the `n`-th position
- the functions that have `char c='\n'` as a parameter, read until the specified *delimiter* is read in;
  the examples here use newline (`\n`), but any character is okay to use

files.

- file handling involves three steps:
  1. opening the file (for reading or writing)
  2. reading from or writing to the file
  3. closing the file
- files in C++ are *sequential access*
- think of a cursor that sits at a position in the file;
  with each read and write operation, you move that cursor's position in the file
- the last position in the file is called the "end-of-file", which is typically abbreviated as `eof`
- all the functions described on the next few slides are defined in the either the `<ifstream>` header file (for files you want to read from) or the `<ofstream>` header file (for files you want to write to)

opening a file for reading.

- first you have to define a variable of type `ifstream`;
  this "input file" variable will act like the cursor in the file and will point sequentially from one character in the file to the next, as you read characters from the file
- then you have to open the file:

```
ifstream inFile; // declare input file variable
inFile.open( "myfile.dat", ios::in ); // open the file
```

- you should check to make sure the file was opened successfully;
  if it was, then `inFile` will be assigned a number greater than 0;
  if there was an error, then `inFile` will be set to 0, which can also be evaluated as the boolean value `false`; so you can test like this:

```
if ( ! inFile ) {
  cout << "error opening input file!\n";  // output error message
  exit( 1 );  // exit the program
}
```

- note that the method `ifstream.open()` takes two arguments:
  - `filename`: a string containing the name of the file you want to open; this file is in the current working directory or else you have to include a full path specification
  - `mode`: which is set to `ios::in` when opening a file for input

reading from a file.

- once the file is open, you can read from it
- you read from it in almost the same way that you read from the keyboard
- when you read from the keyboard, you use `cin >> ...`
- when you read from your input file, you use `inFile >> ...`
- here is an example:

```
int x, y;
inFile >> x;
inFile >> y;
```

- here is another example:

```
int x, y;
inFile >> x >> y;
```

- when reading from a file, you will need to check to make sure you have not read past the end of the file;
  you do this by calling:
  `inFile.eof()`
  which will:
  — return `true` when you have gotten to the end of the file (i.e., read everything in the file)
  — return `false` when there is still something to read inside the file
- for example:

```
while ( ! inFile.eof() ) {
  inFile >> x;
  cout << "x = " << x << endl;
} // end of while loop
```

opening a file for writing.

- first you have to define a variable of type `ofstream`;
  this “output file” variable will act like the cursor in the file and will point to the end of the file, advancing as you write characters to the file
- then you have to open the file:

```
ofstream outFile; // declare output file variable
outFile.open( "myfile.dat", ios::out ); // open the file
```

- you should check to make sure the file was opened successfully; if it was, then `outFile` will be assigned a number greater than 0; if there was an error, then `outFile` will be set to 0, which can also be evaluated as the boolean value `false`; so you can test like this:

```
if ( ! outFile ) {
  cout << "error opening output file!\n";  // output error message
  exit( 1 );  // exit the program
}
```

- note that the method `ofstream.open()` takes two arguments:
  - `filename`: a string containing the name of the file you want to open; this file is in the current working directory or else you have to include a full path specification
  - `mode`: which is set to `ios::out` when opening a file for output

writing to a file.

- once the file is open, you can write to it
- you write to it in almost the same way that you write to the screen
- when you write to the screen, you use `cout << ...`
- when you write to your output file, you use `outFile << ...`
- here is an example:

```
outFile << "hello world!\n";
```

- here is another example:

```
int x;
outFile << "x = " << x << endl;
```

closing a file.

- when you are done reading from or writing to a file, you need to close the file
- you do this using the `close()` function, which is part of both `ifstream` and `ofstream`
- so, to close a file that you opened for reading, you have do this, e.g.:

```
inFile.close(); // close input file
```

- and, to close a file that you opened for writing, you have do this, e.g.:

```
outFile.close(); // close output file
```

- that’s all!

using strings as streams.

- you can also use a string as a stream (i.e., to write output to a string or read input from a string)
- the `sstream` header contains two data types:
  `ostringstream` for output
  `istringstream` for input
- example:

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

int main() {

#define MAXBUF 10
  char buf[MAXBUF];
  char c;
  istringstream instring( "my test string" );
```

```
  ostringstream outstring;
  ostringstream outstring2( buf,ios::app );

  // input is read from "instring"
  instring >> c;
  cout << "c=[" << c << "]\n";

  // output is written to "outstring" and "outstring2"
  outstring << c;
  outstring << c;
  cout << "outstring=[" << outstring.str() << "]\n";

  strcpy( buf,"" );
  outstring2 << 'A';
  outstring2 << 'B';
  outstring2 << 'C';
  outstring2 << "DEF";
  cout << "outstring2=[" << outstring2.str() << "]\n";
}
```

```
SAMPLE OUTPUT:

c=[m]
outstring=[mm]
outstring2=[ABCDEF]
```

ctype functions and macros (1).

- character handling library

  `#include <ctype.h>`
- digit recognition functions (bases 10 and 16)
- alphanumeric character recognition
- case recognition/conversion
- character type recognition
- these are all of the form:

  `int isdigit( int c );`

  where the argument `c` is declared as an `int`, but it is intepreted as a `char`
  so if `c = '0'` (i.e., the ASCII value '0', index=48), then the function returns *true* (non-zero int)
  but if `c = 0` (i.e., the ASCII value NULL, index=0), then the function returns *false* (0)

ctype functions and macros (2).

digit recognition functions (bases 10 and 16)

- `int isdigit( int c );`

  returns *true* (i.e., non-zero int) if c is a decimal digit (i.e., in the range `'0'..'9'`); returns 0 otherwise

- `int isxdigit( int c );`

  returns *true* (i.e., non-zero int) if c is a hexadecimal digit (i.e., in the range `'0'..'9','A'..'F'`); returns 0 otherwise

ctype functions and macros (3).

alphanumeric character recognition

- `int isalpha( int c );`

  returns *true* (i.e., non-zero int) if c is a letter (i.e., in the range `'A'..'Z','a'..'z'`); returns 0 otherwise

- `int isalnum( int c );`

  returns *true* (i.e., non-zero int) if c is an alphanumeric character (i.e., in the range `'A'..'Z','a'..'z','0'..'9'`); returns 0 otherwise

ctype functions and macros (4).

case recognition

- `int islower( int c );`

  returns *true* (i.e., non-zero int) if c is a lowercase letter (i.e., in the range `'a'..'z'`); returns 0 otherwise

- `int isupper( int c );`

  returns *true* (i.e., non-zero int) if c is an uppercase letter (i.e., in the range `'A'..'Z'`); returns 0 otherwise

case conversion

- `int tolower( int c );`

  returns the value of c converted to a lowercase letter (does nothing if c is not a letter or if c is already lowercase)

- `int toupper( int c );`

  returns the value of c converted to an uppercase letter (does nothing if c is not a letter or if c is already uppercase)

ctype functions and macros (5).

character type recognition

- `int isspace( int c );`

  returns *true* (i.e., non-zero int) if c is a space; returns 0 otherwise

- `int iscntrl( int c );`

  returns *true* (i.e., non-zero int) if c is a control character; returns 0 otherwise

- `int ispunct( int c );`

  returns *true* (i.e., non-zero int) if c is a punctuation mark; returns 0 otherwise

- `int isprint( int c );`

  returns *true* (i.e., non-zero int) if c is a printable character; returns 0 otherwise

- `int isgraph( int c );`

  returns *true* (i.e., non-zero int) if c is a graphics character; returns 0 otherwise

C style I/O (1).

- `#include <stdio.h>`
  OR
  `#include <cstdio> using namespace std;`
- `int printf(const char *format, ...)` formatted output to stdout
- formatting:

conversion character	argument	description
c	char	prints a single character
d or i	int	prints an integer
u	int	prints an unsigned int
o	int	prints an integer in octal
x or X	int	prints an integer in hexadecimal
e or E	float or double	print in scientific notation
f	float or double	print floating point value
g or G	float or double	same as e,E,f, or f — whichever uses fewest characters
s	char*	print a string
p	void*	print a pointer
%	none	print the % character

- note that there is also `sprintf()`, which is like the C++ `ostringstream` where you can write output to a string

C style I/O (2).

- some flags:

flag	description
-	left justify
+	print plus or minus sign
0	print leading zeros (instead of spaces)

- also specify field width and precision
- example:

```
printf( "i=%d s=%d f=6.3f m=43s",i,s,f,m );
```

C style I/O (3).

- `int scanf(const char *format, ...)` formatted output to stdout
- formatting:

conversion character	argument	description
c	char*	reads a single character
d	int*	reads a decimal integer
i	int*	reads an integer in decimal, octal (leading 0) or hex (leading 0x)
u	int*	reads an unsigned int
o	int*	reads an integer in octal
x or X	int*	reads an integer in hexadecimal
e, E, f, F, g or G	float or double	reads a floating point value
s	char*	reads a string
p	void**	reads a pointer

- note that there is also `sscanf()`, which is like the C++ `istringstream` where you can read input from string

stdio example.

```
#include <stdio.h>

void main( void ) {
  int n = 0; /* initialization required */
  printf( "how much wood could a woodchuck chuck\n" );
  printf( "if a woodchuck could chuck wood?" ); /* prompt user */
  scanf( "%d",&n ); /* read input */
  printf( "the woodchuck can chuck %d pieces of wood!\n",n );
  return;
}
```

```
$ ./a.out
how much wood could a woodchuck chuck
if a woodchuck could chuck wood? 12345
the woodchuck can chuck 12345 pieces of wood!
```