Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 11

- C-Strings
- Unix I/O
- C I/O

C-Strings: Constants

A C-string is a sequence of characters

C-string constants are double-quoted

- `cout << "I am a string" << endl;`
- `cout << "hello world\n";`
- can contain escape sequences such as `\n` or `\a`
- `"` in the text is represented by `\"`
  e.g. `cout << "\"";`

C-String Representation

```
char s[9] = "Hello!";

s[0] s[1] s[2] s[3] s[4] s[5] s[6] s[7] s[8]
  H    e    l    l    o    !   \0  <both undef>

char s[] = "Hello!"; // reserves enough space to
                     // hold Hello! + \0
-> sizeof(s) = 7
```

- Array of characters which contains the character sequence
- Plus end-marker '\0' (0 byte)
- Some operations inefficient! C++ comes with a more sophisticated string template class (later)
- C-strings can be initialized via =

C-String Pitfalls

- Ensure that the char array is big enough - must hold characters + end-marker 0!
- Character with code 0 cannot be represented in a C-string because 0 indicates end-of-string
- Assignments other than initializations are illegal
- == and other relational operators don't work with C-strings
- Does not sound very useful
- Solution: library functions!

C-String Library Functions (<cstring> resp. <string.h>)

```
int strlen(const char s[]);
```

- returns the # of characters in s excluding the end-marker

```
void strcpy(char dest[], const char src[]);
```

- copies string src to dest
  (dest must be large enough!)

```
void strcat(char dest[], const char src[]);
```

- appends string src to dest overwriting its end-marker and adds '\0'
  (dest must be large enough!)

C-String Library Functions (2)

```
int strcmp(const char s1[], const char s2[]);
```

- compares strings s1 and s2
- returns 0 iff they are equal
- return value > 0 iff s1 $>$ s2 (lexicographical order)
- return value < 0 iff s1 $<$ s2

```
char *strdup(const char *s);
```

- returns pointer to copy of string s
- string is allocated using `malloc()`
- if no longer needed, delete via `free(s);`

To learn more about functions in <string.h> issue
`man string.h`

C-String Assignment

```
#include <cstring>

char s1[] = "hello";
char s2[100];

strcpy(s2, s1);  // s2 equals "hello"

char s_too_short[2];

strcpy(s_too_short, s1); // undefined!
```


C-String Comparison

```
char a[] = "aaa";
char b[] = "aaaa";
char c[] = "b";

strcmp(a, a) == 0
strcmp(a, b) < 0
strcmp(c, b) > 0

strlen(b) == 4
```

strlen & strcpy Implementation

```
// return length of string, pointer version
int strlen(const char *s)
{
  const char *p = s;
  while (*p) ++p;
  return p-s; // pointer arithmetic
}


// copy t to s, pointer version
void strcpy(char *s, const char *t)
{
  while (*s++ = *t++);
}
```


strcat Implementation

```
// appends the src string to the dest string
// overwriting the '\0' character at the end of
// dest and then adds a terminating '\0' character

void strcat(char dest[], const char src[])
{
  int i=0;
  while (dest[i]) ++i;  // find end-marker
  int j=0;
  char c;
  do {
    c = dest[i++] = src[j++]; // append src
  } while (c);
}
```


C-Strings & C++ I/O

Output using << operator works. E.g.

```
char s[] = "hello";  cout << s;
```

Input using >> also possible, BUT

- leading whitespaces (blanks, tabs, newline) are skipped
- reading stops at next whitespace
- string length in input may be larger than string variable! Unsafe!
  DON'T USE >> WITH C-STRINGS!

Better Solution

```
const int N = 5;
char s[N];

cin.getline(s, N); cout << s << endl;

Input:  123456789\n    Output: 1234
```

- Input stream function
  `void getline(char s[], int max_total_len);`
- Reads entire input line into C-string s including whitespaces
- Copies up to `max_total_len-1` characters
- End-of-line character ('\n') is not copied
- Even better: C++ string class (later)

Command-Line Parameters

```
// print all command-line arguments
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  for (int i=0; i < argc; ++i)
    cout << "arg-" << i << " \"" << argv[i] << '"' << endl;
}
```

prototype: `int main(int argc, char *argv[]);`

- `argc`: number of command-line arguments
- `argv`: array of pointers to command-line args.
- `argv[0]`: pointer to program name
- `argv[1]`: pointer to first argument, ...

Example

```
./foo -o x "foo bar" 'moe'

output:

arg-0 "./foo"
arg-1 "-o"
arg-2 "x"
arg-3 "foo bar"
arg-4 "moe"
```

- When invoking a command, shell cuts input line into pieces
- uses space (' ') as delimiter (but obeys strings)
- removes leading and trailing spaces

typedef

```
typedef signed char   sint1;
typedef unsigned char uint1;
typedef signed int    sint4;
typedef float real;
// typedef double real; // alternative!
sint4 i;  // signed four-byte integer
uint1 c;  // unsigned one-byte integer
real  r;  // float or double

typedef const char *ccptr;
int strlen(ccptr s) { ... }
```

- Type aliases are new type names for existing types
- Syntax: `typedef` <variable-declaration>;
- Variable identifier is treated as type name
- Increases readability and portability
- Can simplify complex type expressions

void*

Generic C pointer type (similar to Java Object reference)

Any pointer can be assigned to a void* pointer

void* pointers cannot be dereferenced (must use cast)

Used for creating generic containers and when actual pointer type does not matter:

```
// in <cstring>
// copies n bytes from src to dest (non-overlapping)
void *memcpy(void *dest, const void *src, size_t n);

float a[N], b[N];

memcpy(b, a, sizeof(a));          // copies a to b
                // void* parameters => no cast needed
```

Unix I/O

- In Unix all input and output is done by reading or writing to files
- All devices are files (`/dev/...`) with special I/O semantics
- Open file before using it
  - System checks access permissions
  - If OK, it returns a small non-negative number – the file descriptor
- File descriptors (fd) 0,1,2 are called standard input, standard output, and standard error, resp.
- C file pointers (fd wrappers) : stdin, stdout, stderr
- C++ file streams cin, cout, cerr

Redirection

- The command shell connects fd 0,1,2 with the console (input: keyboard, output: text window)
- User can redirect I/O to and from files using > , >>, and <
- >> appends output to a file
- `./prog < infile > outfile`
  connects file desriptors 0 and 1 to the named files
- Normally file descriptor 2 remains attached to the console to display error messages
- Can also be redirected: syntax is shell-dependent, e.g. bash: `./prog > xxx  2>&1`
  both stdout and stderr are redirected

Low-level C I/O

- Low-level I/O is handled by library functions
  - `open, creat, read, write, close`
  - `write(1, "hello world", strlen("hello world"));`
  - first argument is file descriptor (1 = stdout)
- fds 0,1,2 are opened when program starts
- All other files have to be opened
  - `int open(char *name, int flags, int perms)`
  - file name, access flags, access permissions
  - `int fd = open("foo", O_WRONLY, 0666); //ugo+rw`
  - error iff return value < 0
- `man 2 open/read/write...`

Example

```
// write one million integers to a file in binary format
#include <cstdio>
#include <cstdlib>
#include <fcntl.h>
#include <unistd.h>

int main()
{
  const int N = 1000000;
  int *a = new int[N];
  for (int i=0; i < N; ++i) a[i] = i;

  int fd = open("data", O_WRONLY);
  if (fd < 0) {
    perror("encountered error"); exit(10);
  }
  if (write(fd, a, N*sizeof(a[0])) < 0) {
    perror("encountered error"); exit(10);
  }
  if (close(fd) < 0) {
    perror("encountered error"); exit(10);
  }
}
```

Wrapper struct FILE

- <cstdio> provides a wrapper for the low-level I/O routines: struct FILE
- more convenient
- buffered: data is not transferred to device/file immediately. It's appended to buffer which gets written when full ⇝ faster!


```
// C library version / SHOULD CHECK FOR I/O ERRORS!
#include <cstdio>
#include <cstdlib>

int main()
{
  FILE *fp = fopen("foo", "w");
  if (!fp) { fprintf(stderr, "error"); exit(10); }
  for (int i=0; i < 500000; ++i) fprintf(fp, "%d ", i);
  fclose(fp);
}

// C++ library version / SHOULD CHECK FOR I/O ERRORS!
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
  ofstream of("foo");
  if (!of) { cerr << "error"; exit(10); }
  for (int i=0; i < 500000; ++i) of << i << " ";
  // of.close();  not needed - closed when of is destroyed
}

// C++ version ~1.25 times slower, but typesafe and extensible
```


FILE Functions (1)

```
FILE *fopen(char *filename, char *mode);
```

- returns 0 if something went wrong (errno contains error code)
- modes:
  "r": read "r+": read & write
  "w": write (truncate) "w+": read & write (trunc.)
  "a": append
- `FILE *fp = fopen("foo", "w"); if (!fp) { // error`

```
int fclose(FILE *fp);
```

- closes file, returns 0 iff no error occurred

FILE Functions (2)

```
int fgetc(FILE *fp);
```
reads next character from stream ($\geqslant 0$) or EOF if end-of-file or error

```
int fputc(int c, FILE *fp);
```
writes character c to stream, returns EOF iff error occurred

```
int feof(FILE *fp);
```
$\neq 0$ iff end of file reached

```
int ferror(FILE *fp);
```
$\neq 0$ iff error occurred
global variable errno contains error code (man errno)
perror("Remark"); prints error description