Lecture 26

- Unix & C I/O
- Review: C++ Tips

Final Exam

- Monday, April 26, 2-5pm, here
- Bring **OneCard** - will be checked
- **Closed Book**
- Covered material: **everything**
  - **lectures, labs, homeworks, assignments**

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 0,1,2 are called **standard input**, **standard output**, and **standard error**
  - C file pointers (later): 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 <
  - 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,stderr are redirected
    - tcsh: prog >& xxx # to file xxx
- >> appends output to a file

C File I/O

- Low-level I/O is handled by library functions
  - open, creat, read, write, close
  - e.g. write(1, "hello world", strlen("hello world"));
  - first argument is file descriptor (1 = std.output)
- 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_RDONLY, 0666); //ugo+rw
- #include <cstdio> , man 2 open/read/write...

FILE wrapper

- <cstdio> provides a wrapper for the low-level I/O routines - **FILE** - which is more convenient
- 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

- size_t **fwrite**(void *ptr, size_t size, size_t n, FILE *fp)
  - writes size*n bytes to file *fp starting at address ptr
- size_t **fread**(void *ptr, size_t size, size_t n, FILE *fp)
  - reads size*n bytes from file *fp and stores them at ptr
- fwrite/fread return number of successfully written/read bytes, use **feof** and **ferror** to distinguish end of file and read errors

More FILE Functions

- int **feof**(FILE *fp) : != 0 iff end of file reached
- int **ferror**(FILE *fp) : != 0 iff error occurred
  - global variable errno contains error code
- int **fprintf**(FILE *fp, ccptr format, ...): formatted output
- int **fscanf**(FILE *fp, ccptr format, ...): formatted input

Example

```
// C library version
#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
#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();  closed when of is destroyed
}
// C++ version ~1.5 times slower, but typesafe and
// extensible
```


Formatted Output

- typedef const char *ccptr;
- int **printf**(ccptr format, ...) = fprintf(stdout, format, ...)
- int **fprintf**(FILE *fp, ccptr format, ...)
  - formatted data output
  - variable # of parameters to be printed, must match format string. Modern compilers check that.
  - e.g. printf("%d %d %f\n", i, j, real); prints a two integers and a double value in readable form to stdout

Format String

- %c : character
- %s : C string
- %d : integer number
- %f : double precision floating point number
- %e : -"- , scientific notation
- ... many more: man fprintf
- %% = %
- general:
  - % [flags] [width] [prec] [len-mod] conv-spec


```
// format examples
#include <cstdio>

char c = 'x';
int i = 12345;
float f = 3.1415926535;
char *s = "foo";

printf("%% c=%c i=%d f=%f s=%s", c, i, f, s);
// "% c=x i=12345 f=3.141593 s=foo"

printf("%d TEST", i);       // "12345 TEST"
printf("%8dTEST", i);       // "   12345TEST"
printf("%08dTEST", i);      // "00012345TEST"
printf("%-8dTEST", i);      // "12345   TEST"

printf("%f TEST", f);       // "3.141593 TEST"
printf("%.1f TEST", f);     // "3.1 TEST"
printf("%7.2f TEST", f);    // "   3.14  TEST"
printf("%+13.8f TEST", f); // "  +3.14159274 TEST"

printf("%e\n", f);          // "3.141593e+00"
```

Formatted Input

- int **scanf**(ccptr format, ...) = fscanf(stdin, format, ...)
- int **fscanf**(FILE *fp, ccptr format, ...)
  - formatted data input
  - variable number of pointers to variables to be read, must match format string
  - returns number of successfully read values
  - e.g. fscanf(fp, "%d %d %f", &i, &j, &real); reads two integers and a double value and returns 3 if OK
  - DANGEROUS! Hopefully the compiler find type errors

Input Example

```
#include <cstdio>

int a,b,c,r = scanf("%d %d %d", &a, &b, &c);
if (r != 3)// less than 3 values read from stdin
           // => error
int c = fgetc(stdin);
if (c == EOF) { // end of file reached or error
  if (feof(stdin))  // end of file
  else              // error

char buffer[200];
int r = fgets(buffer, 200, stdin);
if (r == 0) // nothing read or error
```


REVIEW – C/C++ Programming Tips

"Wisdom and beauty form a very rare combination."
(Petronius Arbiter, Satyricon XCIV)

"With great power comes great responsibility."
(Spiderman's Uncle)

- Why C?
  - Code is **FAST**; compiler is **FAST**; often only little slower than hand-written assembly language code
  - **Lingua Franca** of computer science
  - **Portable**. C compilers are available on all systems
  - Compilers/interpreters for new languages are often written in C
- Why C++?
  - C + classes + templates: **FAST** + **convenient**
  - You are still in total **control**, unlike Java or C#

From C to C++

- Use **const** and **inline** instead of **#define**
  - Macros are not typesafe
  - Macros may have **unwanted** side effects. Use inline functions instead! (e.g. #define max(a,b) ((a)>(b)?...))
- Prefer **C++ library I/O** over **C library I/O**
  - C's fprintf and friends are unsafe and not extensible
  - C++ iostream class safe and extensible
  - iostream speed is catching up, **so speed is hardly a reason anymore for choosing C-library I/O**
- Prefer C++-style casts
- Distinguish between pointers and references

Memory Management

- Use the same form in **corresponding calls** to new and delete
  - int *p = new Foo; ... delete p;
  - int *p = new Foo[100] ... delete [] p;
- For each new there **must** be a delete
- **Delete pointer members** in destructors
  - otherwise you are creating memory leaks
- No need for **checking the return value** of new
  - It throws an exception if no memory available
- delete p with p=0 is OK (ignored, no check req.)

The "Big 4"

- Define **copy constructor** and **assignment operator** when memory is dynamically allocated
  - default bit-wise copy is not sufficient in this case
- Make destructors **virtual** in base classes
  - otherwise base class pointers can't call the right destr.
- Have operator= **return reference to *this**
  - for iterated assignments a = b = c ...
- Assign to **all** data members in operator=
- Check for **self assignment** in operator=
  - if (this == &rhs) return *this;

Operators

- **Never** overload && || ,
- Distinguish between **prefix and postfix forms** of ++/--
  - they (should) return different types
  - ++i : returns reference to i
  - i++ : returns value of temporary object (**can be slower!**)
- Be **consistent**. E.g.
  - + += prefix++ postfix++ should have **related semantics**

Class/Function Design (1)

- Guard header files against multiple inclusion
  - #ifndef ClassName_H ...
- Strive for **complete** and **minimal** interfaces
  - **complete**: users can do anything they need to do
  - **minimal**: as few functions as possible, no overlapping
- Minimize compilation **dependencies** between files
  - Consider forward declaration in conjunction with pointers/references to minimize file dependencies
  - class Address;
    class Person { ... Adress *address; ... }
  - No need to #include "Address.h"!

Class/Function Design (2)

- **Avoid data members** in public interfaces
  - use inlined get/set functions - more flexible
- Use **const** whenever possible
- Pass and return objects by **reference**
  - But don't return references to non-existent objects like local variables!
- Avoid returning **writable "handles"** to internal data from const member functions
  - otherwise constant objects can be altered

Inheritance

- Make sure public inheritance models **"is a"**
- **Never redefine** an inherited non-virtual function
  - different results for pBase->f() and pDeriv->f()
- **Never redefine** an inherited default parameter value
  - virtual functions are dynamically bound
  - default parameters are statically bound
- Avoid **casting down** the inheritance hierarchy
  - use virtual functions instead

Exceptions

- Use destructors to prevent resource leaks
  - Say **good-bye** to pointers that manipulate local resources - use **smart pointers**
- Prevent resource leaks in constructors
  - destructors are only called for fully constructed objects
- Prevent exceptions from leaving destructors
  - Exceptions within exceptions **terminate** program
  - Special case: exceptions call destructors ...
- Catch exceptions by **reference**
  - all alternatives create problems

Efficiency

- Choose suitable **data structures** and efficient **algorithms**
- Consider the **80-20** rule
  - ~80% of the resources are used by ~20% of the code
  - **Focus** your optimization efforts by using **profilers**
- Avoid frequent **heap memory** allocation
- Know how to **save space**
  - bits, bytes, unions, home-brewed memory allocators
- Understand **costs** of virtual functions, multiple inheritance, exception handling, and RTTI
- Consider alternative libs. (e.g. iostream vs. stdio)

STL Tips (1)

- Choose your containers wisely
  - sequence/associative, speed, memory consumption
- **Careful** when storing **pointers** in containers
  - if the container owns the objects they have to be destroyed before the container is
  - possibly dangling pointers to vanished objects
  - specify comparison functors
- If **speed** matters, use vectors or hashed associative containers. If speed **really** matters, don't use STL (for now, but STL implementations are becoming faster)

STL Tips (2)

- Make sure **destination ranges** are big enough
- Note which algorithms expect **sorted ranges**
- Have realistic expectations about **thread safety** of STL containers: YOU need to lock containers
- Call empty() instead of checking size() against 0
- Make element copies cheap and correct
  - **STL copies elements often**
- Always have comparison functions return **false** for equal values
- More tips in: **S.Meyers: Effective STL**

Fin, Ende, The End

- I am always looking for **good students**!
  - Design/Implementation of a Real-Time Strategy Game environment:
    - **"Hack-free" server/client operation**
    - **3d Graphics, artificial intelligence**
  - Making machines smarter:
    - **Machine Learning**
    - **Heuristic Search, Planning**
  - **Interested?** See me in December
- Apply for an NSERC Summer Scholarship!