Lecture 26 More on Files CSI333

What is a “mangled symbol”? 

A mangled symbol is a string composed of a C++ function name concatenated with other characters that encode the sequence of the types of its arguments, and, if the function is a member of a class (or struct), the name of the class. (JAVA: also the return type!!)

This information is called the signature of the function.

The --demangle option to the symbol table lister utility nm makes nm print this signature information in C++ form instead of the mangled symbol string.

Mangled symbols enable C++ to have two features not in C:
1. (Partly) Type Safe Linkage for Functions.

2. Overloaded operators and function names (ref: iostream classes example, and Strou. sec. 7.4 for explanation).

file file1.c:
int funny( float x )
{
...
}
Compiled C:
funny:
    .... prolog ...
    .... ...
    jr $ra
Compiled C++:
funny_Ff:
    .... prolog ...
    .... ...
    jr $ra

file file2.c:
extern int funny( int )
main()
{
    funny( 38 );
}
Compiled C:
main: .... prolog ...
    jal funny
    jr $ra
Compiled C++:
main: .... prolog ...
    jal funny_Ff
    jr $ra
These illustrate BAD C programming!
gcc -c file1.c -> file1.o OK!
gcc -c file2.c -> file2.o OK!
LINK STEP:
gcc file1.o file2.o -o buggy -> buggy
DOESN’T REPORT AN ERROR!, but, file2’s code
puts the 2-s comp. binary integer representation of
38 in the 1st argument register, but file1’s code
expects to find an single precision IEEE represented
value in the 1st argument register. The floating point
rep. NOT be for mathematical value 38!

With C++’s type safe linkage, the LINK step
g++ file1.o file2.o -o buggy
will report an ERROR and FAIL:
**file2.o has undefined symbol funny_Fi**
since symbols funny_Fi and funny_Ff are different
symbols.

(Pitfall: g++ mangled symbols do NOT express the
RETURN TYPE.)

Conclusion: NEVER copy declarations into .cpp files!
Always **include** a common header file in the
implementation file and every file that **uses** the
function.
A Few Ways to Format Files:

1. ASCII file of standard integer, floating point, string or character tokens separated by whitespace; lines of such tokens.

2. Array of fixed length “binary” data representations. Example: TMIPS binary file reflects the bytes in long unsigned int MEMORY[256];

3. ASCII file of fixed length records. When printed, it looks like a table.

4. Array of fixed length “binary” representations of C structures.

Read the whole array of unsigned ints:

```cpp
ifstream MyFile( /* filename here */ );
MyFile.read( MEMORY, sizeof( MEMORY ) );
if( MyFile.gcount() != sizeof( MEMORY ) )
{ /* Something’s wrong... */ }
```

Read one integer:

```cpp
long unsigned int OneInt;
MyFile.seekg( 5*sizeof(long unsigned int),
             ios::beg );
MyFile.read( &OneInt,
             1*sizeof( long unsigned int ) );
assert( MEMORY[5] == OneInt );
// CRASH if something is wrong.
```
(A long unsigned int is 4 bytes nowadays.)

Every 4 bytes of this “binary” file comprise one long int.
The bytes are arranged in the FILE according to the BYTE ORDER of the machine.

-------------

/******* From /usr/include/sys/elf.h ***********/

#define EI_NIDENT 16

typedef struct {
  unsigned char e_ident[ EI_NIDENT ]; /* ident bytes */
  Elf32_Half e_type;          /* file type */
  Elf32_Half e_machine;       /* target machine */
  Elf32_Word e_version;       /* file version */
  Elf32_Addr e_entry;         /* start address */
  Elf32_Off e_phoff;          /* phdr file offset */
  Elf32_Off e_shoff;          /* shdr file offset */
  Elf32_Word e_flags;         /* file flags */
  Elf32_Half e_ehsize;        /* sizeof ehdr */
  Elf32_Half e_phentsize;     /* sizeof phdr */
  Elf32_Half e_phnum;         /* number phdrs */
  Elf32_Half e_shentsize;     /* sizeof shdr */
  Elf32_Half e_shnum;         /* number shdrrs */
  Elf32_Half e_shstrndx;      /* shdr string index */
} Elf32_Ehdr;
(1) ELF HEADER

ELF header

- e_shoff
- e_shentsize
- e_shstrndx
- e_shnum

section header table

sh_type=SHT_STRTAB
- sh_offset
- sh_size

section header for section name string table.

The section name string table (contains the names of the sections).

Some fundamental types of sections:

- sh_type=SHT_STRTAB  string table
- sh_type=SHT_PROGBITS  executable code or data
- sh_type=SHT_RELA or SHT_REL  relocation (modification) records
To load segments, extract:

1. ELF HEADER
2. Program header table
3. Loadable segments

Program (segment) header table

- p_type: PT_LOAD
- p_offset
- p_vaddr
- p_filesz
- p_memsz
- p_align
- p_flags: PF_X | PF_W | PF_R

ELF File

Program (segment) in ELF File

Virtual Memory

Segment installed in Virtual Memory

Program header table

ELF header

- e_ident: 0x7f ELF...
- e_type
- e_machine
- e_version
- e_entry: START_VADDR
- e_phoff
- e_phentsize
- e_phnum

Note: The diagram illustrates the structure of an ELF (Execution-Linkable Format) file, detailing how to extract segments for loading into virtual memory.
// This program demonstrates how to setup and load simulated
// memory segments from a REAL executable file from System V
// Unix systems!
#include <fstream.h>
#include <iomanip.h>
#include "elf.h"

int main( int argc, char * argv[] )
{
    Elf32_Ehdr myheader;
    if( argc < 2 )
    {
        cerr << "Missing filename argument.\n";
        exit( 1 );
    }
    ifstream ELFfile( argv[ 1 ] );
    ELFfile.read( &myheader, sizeof( myheader ) );

    Elf32_Phdr * psegmentTable = new Elf32_Phdr[myheader.e_phnum];
    // get space in which to read the (program) segment header TABLE.

    ELFfile.seekg( myheader.e_phoff, ios::beg );
    // seek to where the table appears in the ELF file.

    long int segmentTableSize = myheader.e_phentsize * myheader.e_phnum;
    // Calculate how many bytes to read from the ELF file.

    ELFfile.read( psegmentTable, segmentTableSize );

    for( int i = 0; i < myheader.e_phnum; i++ )
    {
        if( psegmentTable[i].p_type == PT_LOAD )
        {
            char * pmemory = new char[ psegmentTable[i].p_memsz ];
            // allocate C++ memory

            ELFfile.seekg( psegmentTable[i].p_offset, ios::beg );
            // seek to position in ELF file to read.

            ELFfile.read( pmemory, psegmentTable[i].p_filesz );
            // initialize memory from the file
for( int j = psegmentTable[i].p_filesz; 
j < psegmentTable[i].p_memsz;
j++ )
  pmemory[ j ] = 0; // initialize memory NOT already
  // initialized from the ELF file.

  long unsigned int sim_addr = psegmentTable[i].p_vaddr;
  long unsigned int nwords =
    psegmentTable[i].p_memsz/sizeof( long unsigned int );
  // save pwords, nwords and sim_addr for later use
}
{
  // DO SOME SIMULATION
  return 0;
}
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
static void usage(void)
{
    cout << "buffdemo [--inner n] [--outer n]" << endl;
    exit (1);
}

int main(int argc, char *argv[])
{
    long unsigned int InnerCount = 10000000; //set default
    long unsigned int OuterCount = 20; //set default
    bool doflush = false; //set default
    while( --argc )
    {
        // argc == number of unprocessed arguments beyond argv[0]
        if( !strcmp( *++argv, "--inner") )
        {
            if( !(--argc) ) usage();
            InnerCount = strtol( *++argv, 0, 10 );
        }
        else if( !strcmp( *(argv), "--outer") )
        {
            if( !(--argc) ) usage();
            OuterCount = strtol( *++argv, 0, 10 );
        }
        else if( !strcmp( *(argv), "--flush") )
        {
            doflush = true;
        }
        else
        {
            usage();
        }
    }
    cout << "Outer Loop will count down from " << OuterCount << ":" << endl;
    cout << " Inner loop takes " << InnerCount << " decr/test steps." << endl;
    long unsigned int inner,outer;
    const char BEL = '\x07';
    const char SPACE = ' ';
    for( outer=OuterCount; outer > 0; outer-- )
    {
        for( inner=InnerCount; inner > 0; inner-- )
        {
            /* DO NOTHING */
            cout << SPACE << BEL << outer;
            if(doflush) cout << flush;
        }
        cout << "\nFinished. Flush now.\n" << flush;
    } return 0;
}