Debugging with the gdb debugger

- **Compiling your program.** For C/C++ programs, use the `-g` `-gstabs+` option for the gcc compiler.
- **Using the debugger.**
  - Starting your program under the debugger. Specifying command line arguments.
  - Stopping your program on specified places and conditions. Setting breakpoints, setting conditional breakpoints, watching variables etc.
  - Stepping through a program: instruction at a time, line at a time, over functions etc.
  - Examining what has happened, when your program has stopped. Looking at the stack frames, values of variables etc.
  - Modifying variables in your program.
  - Attaching the debugger to a program that is already running!
- **Documentation.**
  - The gdb debugger has extensive on-line help that can be accessed by typing in `help` at the gdb prompt.
  - A two page reference card is available. (Check Amit’s home page in the section *Handouts for Students*).
  - The complete reference manual is available in HTML.
GDB: Sample Sessions

- **gdb/session0.** Shows how to access built-in help from inside gdb.
- **gdb/session1.** Shows basic usage. Shows how to examine arrays.
- **gdb/session2.** Shows how to examine the stack trace after a segmentation fault.
- **gdb/session3.** Shows how to examine the stack trace from a core file that was dumped after program crashed.
- **gdb/session4.** Shows the usage of breakpoints.
- **gdb/session5.** Shows how to stop at a breakpoint only if certain condition is true. Also shows how to look at structures and manipulate pointers in the debugger.
- **gdb/session6.** Shows how to attach to an already running process to debug it.
Other tools

- **Data Display Debugger (ddd)**. A graphical front-end for gdb. Nice displays of pointers and structures!

- **Eclipse CDT**. An Integrated Development Environment (IDE) for C/C++ that is a plugin for Eclipse. Can be used under Linux, MS Windows or Mac OS.

- **Valgrind**. Valgrind is an open source system for debugging and profiling x86-Linux programs. With the tools that come with Valgrind, you can automatically detect many memory management and threading bugs, avoiding hours of frustrating bug-hunting, making your programs more stable. You can also perform detailed profiling, to speed up and reduce memory use of your programs. Website: [http://valgrind.kde.org/](http://valgrind.kde.org/).
The type void * is the proper type for a generic pointer in C (similar in concept to Object type in Java). Pointers to any type can be assigned to a variable of type void *, which allows polymorphism. The following example shows a polymorphic quicksort function.

```c
void swap(void v[], int i, int j) 
{
    void *tmp = v[i]; v[i] = v[j]; v[j] = tmp;
}

/* qsort: sort v[left]..v[right] into increasing order */
void qsort(void v[], int left, int right,
          int (*compare)(void *, void *))
{
    int i, last;

    if (left >= right) /* do nothing if array contains */
        return; /* less than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i<= right; i++)
        if (((*compare)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, compare);
    qsort(v, last+1, right, compare);
}
```
A Generic Doubly-Linked List

In order to create a generic doubly-linked list, we have a List class that uses a Node class but now each node contains a `void *` pointer to an generic object that will be stored in our list.

For this to work, we need the user to pass us three function pointers: one for obtaining the key value for a given object, another for freeing the object and also one for converting the object into a string representation suitable for printing. We will store these function pointers in the List structure. The user will pass these with the `createList` function.

```c
struct list {
    int size;
    NodePtr head;
    NodePtr tail;
    unsigned long int (*getKey)(void *);
    char * (*toString)(void *);
    void (*freeObject)(void *);
};

/* constructor */
ListPtr createList(unsigned long int(*getKey)(void *),
                    char * (*toString)(void *),
                    void (*freeObject)(void *));
```
/ C-examples/doublyLinkedLists/library/List.h*/
 ifndef __LIST_H
 define __LIST_H
 include <stdio.h>
 include <stdlib.h>
 include "common.h"
 include "Node.h"
 typedef struct list List;
 typedef struct list * ListPtr;

 struct list {
   int size;
   NodePtr head;
   NodePtr tail;
   unsigned long int (*getKey)(void *);
   char * (*toString)(void *);
   void (*freeObject)(void *);
};

 void freeList(ListPtr L); /* destructor */
 int getSize(ListPtr L);
 Boolean isEmpty(ListPtr L);
 void addAtFront(ListPtr list, ... list, NodePtr node);
 NodePtr search(ListPtr list, int key);
 void reverseList(ListPtr L);
 void printList(ListPtr L);

 #endif /* __LIST_H */
#ifndef __NODE_H
#define __NODE_H
/* C-examples/doublyLinkedLists/library/Node.h */

#include <stdio.h>
#include <stdlib.h>
#include "common.h"

typedef struct node Node;
typedef struct node * NodePtr;

struct node {
    void *obj;
    NodePtr next;
    NodePtr prev;
};

NodePtr createNode (void *obj);
void freeNode(NodePtr node, void (*freeObject)(void *));

#endif /* __NODE_H */
```c
ListPtr createList(unsigned long int(*getKey)(void *),
                    char (*toString)(void *),
                    void (*freeObject)(void *))
{
    ListPtr list;
    list = (ListPtr) malloc(sizeof(List));
    list->size = 0;
    list->head = NULL;
    list->tail = NULL;
    list->getKey = getKey;
    list->toString = toString;
    list->freeObject = freeObject;
    return list;
}

/* other functions unchanged from earlier example */
```
Node.c

/* C-examples/doublyLinkedLists/library/Node.c */
#include "Node.h"

NodePtr createNode(void *obj)
{
    NodePtr newNode = (NodePtr) malloc (sizeof(Node));
    newNode->next = NULL;
    newNode->prev = NULL;
    newNode->obj = obj;
    return newNode;
}

void freeNode (NodePtr node, void (*freeObject)(void *))
{
    if (node == NULL) return;
    (*freeObject)(node->obj);
    free(node);
}
Using a Generic List

In order to use the generic list, the user will have to create an object type that they want to use (can be any type with any name). Then they have to provide the three function pointers for getKey, toString and freeObject to the createList function.

See examples in the following directories:
C-examples/doublyLinkedLists/generic/TestList.c
C-examples/doublyLinkedLists/generic/SimpleTest.c
Variable Argument Lists: Overloading in C

- C allows a function call to have a variable number of arguments with the variable argument list mechanism. For more information see the man page for stdarg.
- Here are some standard function calls that use variable argument lists:
  ```c
  int printf(const char *format, ...);
  int scanf(const char *format, ...);
  int execlp(const char *file, const char *arg, ...);
  ```
- See man stdarg for documentation on using variable argument lists.
- An example of processing a variable argument list, test-varargs.c, is discussed on next frame.
Variable Argument Lists Example

/* C-examples/overloading/test-varargs.c */
#include <stdio.h>
#include <stdarg.h>

void strlist(int n, ...)
{
  va_list ap;
  char *s;

  va_start(ap, n);
  while (1) {
    s = va_arg(ap, char *);
    printf("%s\n", s);
    n--;
    if (n==0) break;
  }
  va_end(ap);
}

int main()
{
  strlist(3, "string1", "string2", "string3");
  strlist(2, "string1", "string3");
}
Creating a Shared Library

Suppose we have three C files: f1.c, f2.c, and f3.c that we want to compile and add into a shared library that we will name mylib.

First, we can compile the C files with the flag -shared to the gcc compiler.

```bash
gcc -Wall -fPIC -c -o f1.o f1.c
gcc -Wall -fPIC -c -o f2.o f2.c
gcc -Wall -fPIC -c -o f3.o f3.c
```

Then we can combine, the three object files into one shared library using the ld linker/loader.

```bash
ld -shared -o libmylib.so f1.o f2.o f3.o
```

At this point, we can write a test program that invokes functions from the library and link it with the shared library.

```bash
gcc test1.c -o test1 libmylib.so
```

Or you can specify the -l option:

```bash
gcc -L. test1.c -o test1 -lmylib
```

where the compiler now searches for libmylib.so in a set of directories. The option -L. tells it to also look in the current directory. The code is in C-examples/libraries directory.
Creating a Static Library

Suppose we have three C files: `f1.c`, `f2.c`, and `f3.c` that we want to compile and add into a shared library that we will name `mylib`.

First, we can compile the C files with the flag `-fPIC` to the `gcc` compiler.

```bash
gcc -Wall -fPIC -c -o f1.o f1.c
gcc -Wall -fPIC -c -o f2.o f2.c
gcc -Wall -fPIC -c -o f3.o f3.c
```

Then we can combine, the three object files into one static library using the `ar` archive program.

```bash
ar rcv libmylib.a f1.o f2.o f3.o
```

At this point, we can write a test program that invokes functions from the library and link it with the static library.

```bash
gcc -Wall -static -I. -L. test1.c -lmylib -o test1.static
```
Shared libraries: dynamic versus static linking

- **Dynamic linking.** This is the default. Here the library is kept separate from the executable, which allows the library code to be shared by more than one executable (or more than one copy of the same executable). Makes the executable size small leading to faster loading up time. The downside is that if we move the executable to another system but the library is missing there, the executable will not run!

- **Static linking.** With static linking we can create a self contained executable. We need to create a static library that we can then link to using the `-static` option with the C compiler. See the output below. Note that static executable are very large in comparison to dynamically linked executables.

```
[amit@dslamit libraries]$ ls -l
-rwxr-xr-x 1 amit home  2503 Aug 30  00:16 libmylib.so
-rwxr-xr-x 1 amit home  5220 Aug 30  00:16 test1
-rwxr-xr-x 1 amit home 404507 Aug 30  00:16 test1.static
```
Using Shared Libraries

- We can use libraries to help organize our own projects as well. For example, we can create a library that contains our list, binary search tree and hash table classes. Then we can link to our library. This saves us from having to copy and include the code for the basic data structures in each program we want to use them. A typical example is to create a directory named `lib` in your home directory. Place your library in this directory. Then add the following line to the end of the file `~/.bashrc`.

  ```bash
  export LD_LIBRARY_PATH= /lib:$LD_LIBRARY_PATH
  ```

  You will also have add the option `-L$(HOME)/lib` when you compile your programs.

- Similarly, we can place all the relevant header files (like `Job.h`, `Node.h`, and `List.h`) in the the directory `include` in your home directory. Then include the option `-I$(HOME)/include` anytime you are compiling a program that uses the List class. You can now include the header files as follows:

  ```c
  #include <Job.h>
  #include <Node.h>
  #include <List.h>
  ```

  The compiler will automatically search for the files in the include directory in your home directory.

- After the above two steps, you can simply link in your library without having to copy the code to each new directory you may want to work in.

  ```bash
  gcc -Wall -I$(HOME)/include -L$(HOME)/lib -o prog1 program1.c -lmylib
  ```
Plugins: What is a plugin?

- A plugin is code that can become part of a program under that program’s control.
- For C and C++ programs plugins are typically implemented as shared libraries that are dynamically loaded (and sometimes unloaded) by the main program.
- Example applications include media players (e.g., an MPEG4 player) for web browsers, specialized filters for image processing applications (e.g., Adobe Photoshop), dynamically loadable drivers for operating systems (e.g., Linux modules) and so forth.
Plugins

Shared libraries represent packages and objects. The system calls `dlclose`, `dlerror`, `dlopen`, `dlsym` provide the programming interface to dynamic linking loader. These calls are Linux specific (and not ANSI C). Equivalent versions are available under other operating systems.

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);
const char *dlerror(void);
void *dlsym(void *handle, char *symbol);
int dlclose(void *handle);
```

See manual page for `dlopen` for more details. Common value of the `flag` argument is `RTLD_LAZY`, which means to resolve undefined symbols as code from the dynamic library is executed.
Plugins

dlopen("/opt/lib/libctest.so", RTLD_LAZY);

Open shared library named "libctest.so". The second argument indicates the binding. Returns NULL if it fails. Options:

- **RTLD_LAZY**: If specified, Linux is not concerned about unresolved symbols until they are referenced.
- **RTLD_NOW**: All unresolved symbols must be resolved when dlopen() is called.
- **RTLD_GLOBAL**: Make symbol libraries visible.
Plugins

dlsym(lib_handle, "ctest1");

Returns address to the function which has been loaded with the shared library. Returns NULL if it fails.

Options:

- dlsym() allows a process to obtain the address of a symbol defined within an object made accessible through a dlopen() call.
- lib_handle is the value returned from a call to dlopen() (and which has not since been released via a call to dlclose())
- name (in this case “ctest1”) is the symbol’s name as a character string. dlsym() will search for the named symbol in all objects loaded automatically as a result of loading the object referenced by handle (see dlopen()).
- Load ordering is used in dlsym() operations upon the global symbol object. The symbol resolution algorithm used will be according to the dependency order (RTLD_LAZY etc.) as described in dlopen().
/* plugin1.c */
 * gcc -Wall -c plugin1.c
 * gcc -shared -lc -o plugin1.so plugin1.o

/* C-examples/plugin/plugin1.c */

#include <stdio.h>
void plugin(void)
{
    printf("This is plug-in 1\n");
}
Plugin: Example (contd.)

/* plugin2.c */
* gcc -Wall -c plugin2.c
* gcc -shared -lc -o plugin2.so plugin2.o
*/

/* C-examples/plugin/plugin1.c */

#include <stdio.h>

void plugin(void)
{
    printf("This is the second plug-in\n");
}

Plugin: Example (contd.)

/* C-examples/plugin/runplug.c */
#define MAX_BUF 1024
#include <stdio.h>
#include <string.h>
/* dll include file */
#include <dlfcn.h>
/* dll variables */
void *handle;  /* handle of shared library */
void (*function)(void);  /* pointer to the plug-in function */
const char *dlError;  /* error string */

int main(int argc, char **argv)
{
    char buf[MAX_BUF];
    char plugName[MAX_BUF];

    while (1)
    {
        /* get plug-in name */
        printf("Enter plugin name (exit to exit): ");
        fgets(buf, MAX_BUF, stdin);
        buf[strlen(buf)-1] = '\0';  /* change \n to \0 */
        sprintf(plugName, "./%s", buf);  /* start from current dir */

        /* checks for exit */
        if (!strcmp(plugName, "./exit"))
            return 0;

        /* open a library */
        handle = dlopen(plugName, RTLD_LAZY);
        if ((dlError = dlerror()))
            printf("Opening Error: %s\n", dlError);
        continue;

        /* loads the plugin function */
        function = dlsym(handle, "plugin");
        if ((dlError = dlerror()))
            printf("Loading Error: %s\n", dlError);

        /* execute the function */
        (*function)();
        if ((dlError = dlerror()))
            printf("Execution Error: %s\n", dlError);

        /* close library */
        dlclose(handle);
        if ((dlError = dlerror()))
            printf("Closing Error: %s\n", dlError);
    }
    exit(0);
}
Using C++ compiler to write C programs

Using the GNU C++ compiler to compile your C programs gives many advantages.

- The C++ compiler performs stricter checks.
- We can declare variables as we go rather than at the beginning of a function (just like Java).
- We can use `new` and `delete` operators for dynamic memory allocation. The `new` operator works the same as in Java. For example:

  ```c
  int *X = new int[100]; /* create an array of 100 int's */
  for (int i=0; i<100; i++)
    X[i] = i;
  delete [] X; /* free the space used by the array X */
  ```

To use the GNU C++ compiler, just replace `gcc` with `g++`. All other common options for `gcc` work the same.
References