Lecture 9. Process Environment

The `main()` Function

The prototype of the `main()` function, as everybody knows, is:

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

Arguments:
- *`argc`* — number of command line arguments
- *`argv`* — an array of pointers to the arguments.

A sample program using command line arguments is the grading program I wrote for the Architecture III class:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
     int sq=atoi(argv[1]);
     int ts=atoi(argv[2]);
     int p1=atoi(argv[3]);
     int p2=atoi(argv[4]);
     int p3=atoi(argv[5]);
     printf("%d\t%d\t%d\t%d\t%d\n", sq, ts, p1, p2, p3);

     float grade=1+sq*0.3+ts+0.2+p1+0.2+p2*0.2+p3+0.1;

     grade=(grade/176.5)*100;

     printf("Grade=%f\n", grade);
return 0;
}
```

A system routine is called before `main()`, and the address of this routine is the start address for the C program. The command line arguments and the environment are passed on to this routine by the kernel.

Process Termination

There are 8 ways to terminate a program, 5 for normal termination…

1. Return from `main()`
2. Call of `exit()`
3. Call of `_exit()` or `_Exit()`
4. Return of the last thread from its start routine
5. Call of `pthread_exit()` from the last thread

…and 3 for abnormal termination:

6. Call of `abort()`
7. Receipt of a signal
8. Response of the last thread to a cancellation request

Thread-related terminations will be discussed later.

The `exit()`, `_exit()` and `atexit()` Functions

These functions terminate a program normally:

```
#include <stdlib.h>

void exit(int status);

void _Exist(int status);
```

and:

```
#include <unistd.h>

void _exit(int status);
```

The integer argument status is called the **exit status**. It is undefined if either:

(a) The argument is missing;
(b) `main()` does a return without a return value;
(c) `main()` is not declared to return an integer (although my compiler won't let me declare a `void main()` anymore).

When exiting `main()`, `exit(0)` and `return(0)` are equivalent.

When a program exits, it can return to the parent process a small amount of information about the cause of termination, using the *exit status*. This is a value between 0 and 255 that the exiting process passes as an argument to `exit()`.

Normally you should use the exit status to report very broad information about success or failure. You can't provide a lot of detail about the reasons for the failure, and most parent processes would not want much detail anyway.

There are conventions for what sorts of status values certain programs should return. The most common convention is simply 0 for success and 1 for failure. Programs that perform comparison use a different convention: they use status 1 to indicate a mismatch, and status 2 to indicate an inability to compare.

A general convention reserves status values 128 and up for special purposes. In particular, the value 128 is used to indicate failure to execute another program in a subprocess.

```
#include <stdio.h>

int atexit(void (*func)(void));
```

The argument is a pointer to a function (the address of a function).

Function Pointers

You can declare a pointer to a function as follows:

```
return_type (*ptr_name)(arg_list_types)=&Function_name;
```

where the function was:

```
return_type Function_name(arg_list_types args);
```

In the program below, the function is `char *Input(void)`, and the pointer to it is `char *(*function)(void)`. You can pass a function pointer as an argument to another function, and you can cause a function call by pointer:

```
return_type return_value=(*ptr_name)(arg_values);
```

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

char *Input(void);
void Success(char *(*function)(void));

int main(void)
{
     char *(*function)(void)=&Input;

     char *result=(*function)();
     printf("%s\n", result);

     Success(function);
return 0;
}

char *Input(void)
{
     char *x="Adriana Wise";
return x;
}

void Success(char *(*function)(void))
{
     char *y=(*function)();
     if (strcmp(y, "Adriana Wise")==0)
          printf("Hello, %s\n", y);
     printf("Good bye, %s\n", y);
}
```

Now, if we wanted to use the `atexit()` function, there's a trick. The type of function pointer it takes is `void (*ptr)(void)` and nothing else. So, in order to call `atexit()`, you should write a **wrapper function** with the required argument list and return type.

In particular, I wrote a wrapper around `char *Input(void)`, with a void return type and a void argument list, called `void Wrapper(void)`:

```
void Wrapper(void)
{
     char *y;
     y=Input();
}
```

The following version of the program will now output the exit status from `Input()`:

```
#include <stdio.h> /* for printf() */
#include <string.h> /* for strcmp() */
#include <stdbool.h> /* for true and false */
#include <stdlib.h> /* for atexit() */

char *Input(void);
void Wrapper(void);
bool Success(char *(*function)(void));

int main(void)
{
     char *(*function)(void)=&Input;
     void (*ptr)(void)=&Wrapper;
     bool status;
     int exit_status;

     char *result=(*function)();
     printf("%s\n", result);

     exit_status=atexit(ptr);
     printf("Exit status: %d\n", exit_status);
     status=Success(function);
return 0;
}

char *Input(void)
{
     char *x="Adriana Wise";
return x;
}

void Wrapper(void)
{
     char *y;
     y=Input();
}

bool Success(char *(*function)(void))
{
     char *y=(*function)();
     if (strcmp(y, "Adriana Wise")==0)
     {
```

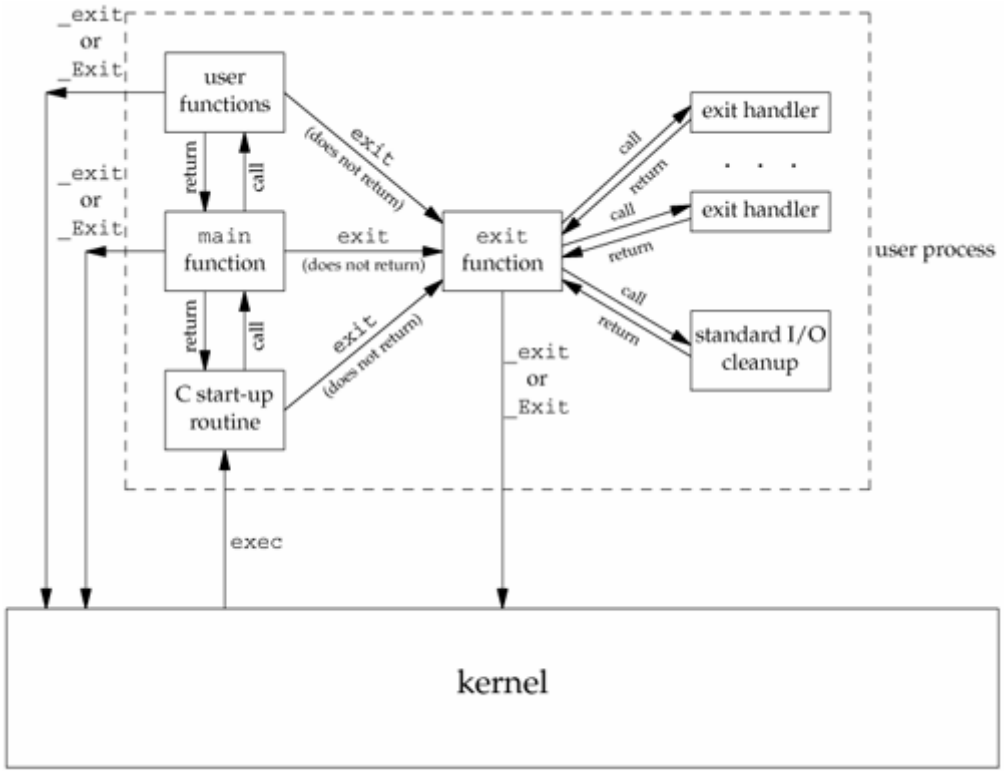
```
            printf("Hello, %s\n", y);
            return true;
      }
      else
      {
            printf("Good bye, %s\n", y);
            return false;
      }
}
```

The output is:

```
$ ./myatexit
Adriana Wise
Exit status: 0
Hello, Adriana Wise
```

The exit status will change, presumably, once the function `Input()` returns with an error condition. Unlikely! We can define the `Success()` function to return with an error if the name is wrong, wrap it, define a pointer to the wrapper, and call it with `atexit()` to check on the status. Can you do that for homework?

The following diagram shows the start-up and termination of a C program:



Exit handlers are user defined routines to deal with an non-zero return value from one of the exit functions. They usually print out error messages.

Here is the example from APUE, slightly modified to fit our fresh understanding of function pointers:

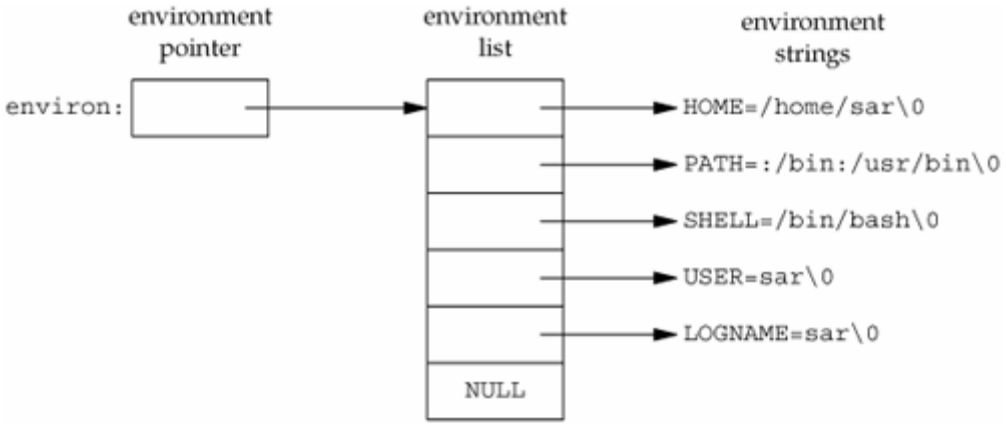
```
#include <stdio.h>
#include <stdlib.h>

void my_exit1(void);
void my_exit2(void);

int main(void)
{
     void (*ptr1)(void)=&my_exit1;
     void (*ptr2)(void)=&my_exit2;

     int status1=atexit(ptr1);
     int status2=atexit(ptr2);

     if (status2!=0)
          printf("Uh-oh. Can't register my_exit2().\n");
     if (status1!=0)
          printf("Grrr. Can't register my_exit1(), either.\n");
     printf("End of main().\n");
return(0);
}

void my_exit1(void)
{
     printf("My 1st exit handler.\n");
}

void my_exit2(void)
{
     printf("My 2nd exit handler.\n");
}
```

The output is:

```
$ ./myexithandlers
End of main().
My 2nd exit handler.
My 1st exit handler.
```

Environment List

Each program is passed an environment list. The environment list is an array of character pointers, each of which point to (contain the start address of) a null-terminated C string. The global variable environ holds this array. The length of environ is variable. Below is an example of such a list:



The following program goes through the environment list and prints out its entries:

```
#include <stdio.h>

int main(void)
{
     extern char **environ;

     char **iterator;
     for (iterator=&environ[0]; *iterator!=NULL; iterator++)
     {
          printf("%s\n", *iterator);
     }
return 0;
}
```

The output is:

```
$ ./myenvir
TERM_PROGRAM=Apple_Terminal
TERM=xterm-256color
SHELL=/bin/bash
TMPDIR=/var/folders/pw/_b5mqq752_qfnkg55947grbc0000gn/T/
Apple_PubSub_Socket_Render=/tmp/launch-RhyYo4/Render
```

```
TERM_PROGRAM_VERSION=326
TERM_SESSION_ID=DFA31DD1-24E5-437B-A7F7-DE791D7D6474
USER=awise
SSH_AUTH_SOCK=/tmp/launch-BVqFyO/Listeners
__CF_USER_TEXT_ENCODING=0x1F5:0:0
__CHECKFIX1436934=1
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/texbin
PWD=/Users/awise/Stevens/Lecture9
LANG=en_US.UTF-8
HOME=/Users/awise
SHLVL=1
LOGNAME=awise
_=./myenvir
```

Memory Layout of a C Program

A C program has the following components:

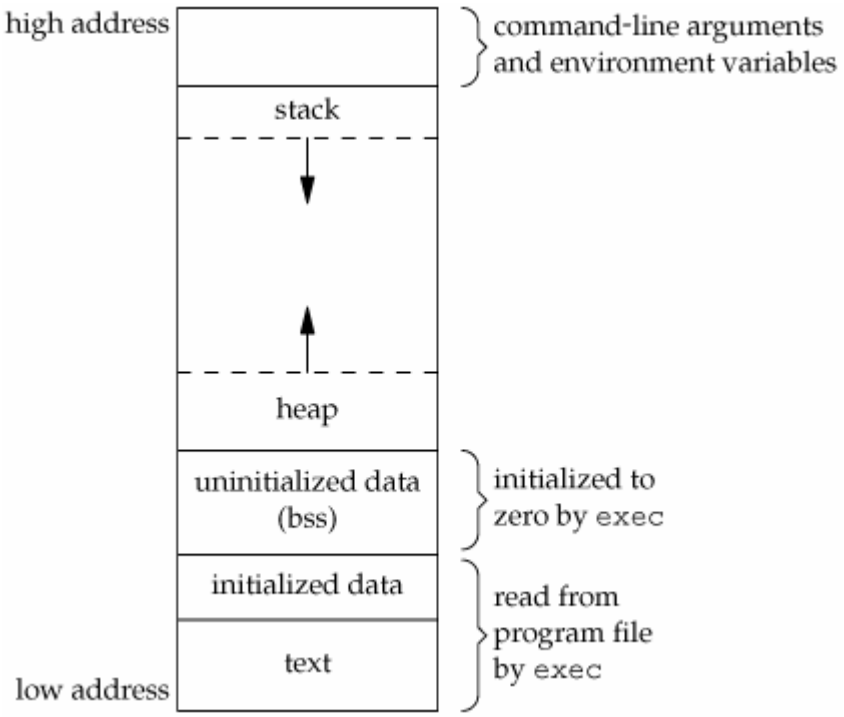
- **text segment** — the machine instructions executed by the CPU. This is shareable, in case frequently executed programs run many instances in parallel (text editors, the C compiler, the shells). Often read-only, to prevent a program from modifying its own instructions;
- **initialized data segment** — or simply "data segment", with variables specifically initialized by the program;
- **uninitialized data segment** — (a.k.a. **bss**, for "block started by symbol") with variables initialized by the kernel (to arithmetic 0 or null pointers) before the program starts to execute (such as malloc-ed arrays);
- **stack** — where automatic variables are stored, and information that is saved each time a function is called (state of the program before the call):
    - the address where to return
    - the caller's environment (state)

The called function allocates room on the stack for its automatic and temporary variables.

Note: This is how *recursive functions* work. Each instance of the function reserves its own stack, so one set of variables doesn't interfere with the variables from another instance of the function.

- **heap** — where dynamic memory allocation takes place, between the uninitialized data and the stack.

The following diagram shows the memory layout of a C program:



The size(1) command reports the sizes (in bytes) of the text, data, and bss segments:

```
$ size /usr/bin/cc /bin/sh
__TEXT    __DATA    __OBJC    others    dec  hex
4096 4096 0    4294975488     4294983680     100004000 /usr/bin/
cc
532480    57344     0    4295008256     4295598080
10009a000 /bin/sh
```

The following program implements a factorial computation using a *recursive function*:

```
#include <stdio.h>

int Factorial(int upperBound);

int main(void)
{
     int n=6;
     int fact=Factorial(n);
     printf("%d!=%d\n", n, fact);
```

```
return 0;
}

int Factorial(int upperBound)
{
     auto int f;
     if (upperBound==0)
     {
          f=1;
     }
     else
     {
          for (int i=1; i<=upperBound; i++)
          {
               f=Factorial(i-1)*i;
          }
     }
return f;
}
```

Shared Libraries

Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine in memory, where all processes can reference it. This reduces the size of the executable file and makes possible the update of the shared libraries without relinking every program that uses them.

Memory Allocation

The following functions are available for memory allocation:

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);
```

All return non-null pointer if OK, `NULL` if error.

```
void free(void *ptr);
```

`malloc()` allocates a specified number of bytes of memory. `calloc()` allocates space for a specified number of objects of a specified size, initialized to all 0s. `realloc()` increases/decreases the size of a previously allocated area, not necessarily contiguously.

The following program illustrates the use of all three functions:

```
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for malloc() */

int main(void)
{
     char **myarray=malloc(2*10*sizeof(char));
     printf("%lu\n", 2*10*sizeof(char));
     myarray[0]="Adriana";
     myarray[1]="Wise";
     char **iterator;
     for (iterator=&myarray[0]; iterator!=&myarray[2]; iterator+
+)
     {
          printf("*%s*\n", *iterator);
     }

     char **myarray2=calloc(2, 10*sizeof(char));
     myarray2[0]="Ludwig van";
     myarray2[1]="Beethoven";
     for      (iterator=&myarray2[0];      iterator!=&myarray2[2];
iterator++)
     {
          printf("*%s*\n", *iterator);
     }

     myarray=realloc(myarray, 2*5*sizeof(char));
     myarray[0]="Felix";
     myarray[1]="Mendelssohn-Bartoldi";
     for (iterator=&myarray[0]; iterator!=&myarray[2]; iterator+
+)
     {
          printf("*%s*\n", myarray[j]);
     }

return 0;
}
```

The output of this program shows the `myarray` variable resized:

```
$ ./mymalloc
20
*Adriana*
*Wise*
*Ludwig van*
*Beethoven*
2
*Felix*
*Mendelssohn-Bartoldi*
```

Environment Variables

The following function returns the values of environment variables, each specified as an argument on each call:

```
#include <stdlib.h>

char *getenv(const char *name);
```

Returns pointer to value associated with name, `NULL` if not set/found.

Here is a small program using `getenv()`:

```
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for gentenv() */

int main(void)
{
     char *value=getenv("HOME");
     printf("%s\n", value);
     printf("%s\n", getenv("LANG"));
     printf("%s\n", getenv("LC_MONETARY"));
return 0;
}
```

…and the output (including a NULL!):

```
$ ./myenvvars
/Users/awise
en_US.UTF-8
(null)
```

The following tables give a comprehensive list of all environment variables across multiple UNIX OSs:

Variable	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Description
LC_COLLATE	•	•	•	•	•	name of locale for collation
LC_CTYPE	•	•	•	•	•	name of locale for character
COLUMNS	•	•	•	•	•	terminal width
LC_MESSAGES	•	•	•	•	•	name of locale for messages
DATEMSK	XSI		•		•	getdate(3) template file
LC_MONETARY	•	•	•	•	•	name of locale for monetary
EDITOR						pathname editing
HOME	•	•	•	•	•	home directory
LC_NUMERIC	•	•	•	•	•	name of locale for numeric
LANG	•	•	•	•	•	editing name of locale
LC_ALL	•	•	•	•	•	name of locale
LC_TIME						name of locale for date/time formatting
LINES	•	•	•	•	•	terminal height
LOGNAME	•	•	•	•	•	login name
MSGVERB	XSI	•			•	fmtmsg(3) message components to process
NLSPATH	XSI	•	•	•	•	sequence of templates for message catalogs
PATH	•	•	•	•	•	list of path prefixes to search for executable file
PWD	•	•	•	•	•	absolute pathname of current working directory
SHELL	•	•	•	•	•	name of user's preferred shell
TERM	•	•	•	•	•	terminal type
TMPDIR	•	•	•	•	•	pathname of directory for creating temporary files
TZ	•	•	•	•	•	time zone information

The following functions add entries to the environment variable list, set values to them, and unset them, respectively:

```
#include <stdlib.h>

int putenv(char *str);

int setenv(const char *name, const char *value, int rewrite);

int unsetenv(const char *name);
```

All return 0 if OK, non-zero if error.

The `setjmp()` and `longjump()` Functions

These provide a replacement for `goto` statements past the reach of an ordinary goto, in particular across branches of the same conditional statement and across functions:

```
#include <setjmp.h>

int setjmp(jmp_buf env);
                                Returns 0 if called directly, non-zero if called after longjmp().
void longjmp(jmp_buf env, int val);
```

Arguments:
env — `jmp_buf` variable (an array) holding state information at the call points, allowing the return to the state of the stack prior to its call, when `longjmp()` is called;
val — integer value that allows more than one call to `longjmp()` for a single call to `setjmp()`

Basically, `setjmp()` is used to save state and `longjmp()` is used to return to that state across multiple nested function calls (mainly to implement error handling functions). The **state** refers to the saved registers a caller function saves on the frame stack when it calls another function. Thus, `setjmp()` saves the contents of registers so `longjmp()` can restore them later: `longjmp()` returns to the state of the program when `setjmp()` was called. This allows branching back through the call frames, to a function that is in the call path of the current function.

The following examples illustrate these functions:

```
#include <stdio.h>
#include <setjmp.h>

int main(int argc, char** argv)
{
     do
     {
          jmp_buf array;
          if (setjmp(array)==0)
          {
                    printf("In if branch.\n");
                    longjmp(array, 1);
```

```
                    printf("I do not appear\n");
          }
          else
          {
                    printf("In else branch!\n");
          }
     } while(0);
return 0;
}
```

…and another one:

```
#include <stdio.h>
#include <setjmp.h>
#include <limits.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>

char *EnterName(void);
char *Success(char *(*ptrEnterName)(void));
void Failure(char *);
char *RemoveEOL(char *string);

char *myname;

int main(int argc, char** argv)
{
     jmp_buf array;
     char *(*ptrEnterName)(void);
     ptrEnterName=&EnterName;

     if (setjmp(array)==0)
     {
               printf("In if branch.\n");
               myname=Success(ptrEnterName);
               printf("In if branch, %s\n", myname);
               longjmp(array, 1);
     }
     else
     {
               printf("In else branch!\n");
               Failure(myname);
     }
return 0;
}
```

```
char *EnterName(void)
{
     printf("Enter name: ");
     char *line=malloc(20*sizeof(char));
     char *name=fgets(line, 20, stdin);
     name=RemoveEOL(name);
     printf("%s\n", name);
return name;
}

char *Success(char *(*ptrEnterName)(void))
{
     char *result=(*ptrEnterName)();
     printf("%s\n", result);
     if (strncmp(result, "Adriana Wise", 20)==0)
     {
          printf("%s is the best\n", result);
     }
return result;
}

void Failure(char *x)
{
     printf("Passed in --->%s<---\n", x);
     int len1=strlen(x);
     int len2=strlen("Adriana Wise");
     printf("%d vs. %d\n", len1, len2);
     if (strncmp(x, "Adriana Wise", strlen("Adriana Wise"))!=0)
          printf("Ooops. You're %s, not Adriana Wise!\n", x);
}

char *RemoveEOL(char *string)
{
     int len=strlen(string);
     if (string[len-1]=='\n')
          string[len-1]='\0';
return string;
}
```

Automatic, Register, and Volatile Variables

An `auto` variable has local variable scope, meaning that it is only "visible" or can only be accessed and changed within the function or block in which it is declared. An `auto` variable disappears when it goes out of scope, freeing the memory which

it was declared with, and any subsequent declaration of the variable will likely take place in a different part of memory.

A `static` variable has global variable scope, in addition to which, when used inside a function, it retains value across consecutive calls of that function. It can be declared either globally or locally (as in our example).

The `volatile` keyword can be used to state that a variable may be changed by hardware, the kernel, another thread etc. For example, the `volatile` keyword may prevent unsafe compiler optimizations for memory-mapped input/output.

The following example attempted to show the effect of volatile, but failed…

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

typedef struct
{
     int command;
     int data;
     int isbusy;
} myGadget;

//void SendCommand(volatile myGadget *g, int command, int data);
void SendCommand(myGadget *g, int command, int data);
static jmp_buf array;

int main(void)
{
     //volatile static myGadget gadget;
     static myGadget gadget;
     gadget.isbusy=0;

     SendCommand(&gadget, 1, 2);
     if (setjmp(array)!=0) //0 if called directly
     {
          printf("After longjmp(): %d\t%d\t%d\n",
               gadget.command, gadget.data, gadget.isbusy);
          exit(0);
     }
     else
     {
          SendCommand(&gadget, 4, 5);
```

```
          printf("Before longjmp(): %d\t%d\t%d\n",
gadget.command, gadget.data, gadget.isbusy);
          longjmp(array, 1);
     }
return 0;
}

//void SendCommand(volatile myGadget *g, int command, int data)
void SendCommand(myGadget *g, int command, int data)
{
     while (g->isbusy)
     {
     }
     g->data=data;
     g->command=command;
}
```

Homework (**due Mar-22-2016**)**:** Write an example program showing the use of `setjmp()` and `longjmp()` across two function calls. You should have two functions, one of which calls the other function. Add and set a string array variable for each function, changing its value. Write a function that prints the string array from within each function. Show what happens to the array upon return from the `longjmp()`.