Good Programming Habits

- More important than debugging: do not write bugs!
- Write simple code!

```c
/* How is anyone supposed to understand this syntax? */
for( ; P("\n") , R--; P(" ") )
for( e=C; e--; P("_" +(*u++)/2) )
P("| " +(u/4)%2);
```

- Always use { } around compounds:

```c
/* This code probably does not do what you expect */
while (!found && i < N)
    found = myok(i);
    i++;
```
Check Function Return Values

- Most functions from the C library return values
  - Most often: >= 0 if everything went fine, < 0 in case of error
- **Always** check these return values!
  - I often don't write it in my slides by lack of space
  - But you do not have any excuse for not doing it...

```c
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd<0) {
    ...
}
```

Use perror()

- There is a standard global variable called **errno**
  - It is defined in `<errno.h>`
- When standard functions fail, they store an error code in **errno**
  - You should look at `errno` for the cause of the problem
- To convert int **errno** into a human-readable string:

```c
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd<0) {
    perror("Error while opening socket");
    exit(1);
}
```
Use Assertions

- Often in a program you know that a given property should normally be true
  - This variable’s value should always be between 0 and 10
  - This pointer should not be null
  - min_data_rate should always be lower than max_data_rate
  - etc...
- Use assert() to check if these properties are true!
  - If the property is true, assert will do nothing
  - Otherwise, it will display a message, stop the program and dump a core
    * Use GDB to read the core file and see what happened!

```c
#include <assert.h>
void assert(scalar expression);
```

Use Assertions

```
cat prog6.c
#include <assert.h>

int main(int argc, char **argv) {
    /* this program should never take any command-line parameter */
    assert(argc==1);

    return 0;
}
$ prog6
$ prog6 wrongparameter
```

$
Avoid These Functions!

- Certain standard C functions do not let you control buffer boundaries
  - **You should never use them!**
  - There is always a good replacement for them

<table>
<thead>
<tr>
<th>Do not use:</th>
<th>Use instead:</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy</td>
<td>strncpy</td>
</tr>
<tr>
<td>sprintf</td>
<td>snprintf</td>
</tr>
<tr>
<td>gets</td>
<td>fgets</td>
</tr>
</tbody>
</table>

Use Proper Formatting

- If you want to display a string:

```c
char string[32];
printf("%s", string); /* This is correct */
printf(string); /* This is WRONG WRONG WRONG */
```

- Try this program (echo):

```c
int main(int argc, char **argv) {
    int i;
    for (i=1;i<argc;i++) { printf(argv[i]); } /* No format string here */
    printf("\n");
}
```

```
$ ./a.out foo
foo
$ ./a.out foo dbaz
foo4195856dbaz
$```
GDB: The GNU Debugger

- A debugger can do two things for you:
  - Run a program step by step, let you follow what it is doing, examine the content of the memory
  - After a program has crashed, load the core file and let you examine what has happened
- GDB can debug programs written in C, C++, Pascal, ADA, etc.
- Current version: 6.6
  - http://www.gnu.org/software/gdb/

Compiling with Debugging Info

- GDB can debug any program
  - But when it executes an instruction, you probably want to see the source code of the instruction being executed
  - This information is normally not present in executable files
- To get them, you must add a flag at compile time
  - This is not necessary at link time (but it cannot hurt)

```
$ gcc -g -c -Wall foo.c
$ gcc -o foo foo.o
$
```

- This includes line-number informations in your compiled programs
GDB Basic Commands

- Basic commands:
  - To run GDB: `gdb [program_name]`
  - To set a breakpoint: `break [function_name]`
    or: `b [function_name]`
    or: `b [filename]:[line_nb]`
  - To display the source around the current instruction: `list` (or: `l`)
  - To start running the program: `run [command-line params]`
  - To continue the execution after a breakpoint: `c`
  - To execute one instruction:
    * `next` or `n` (treats a function call as a single instruction)
    * `step` or `s` (enters inside a function when it is called)
  - To print the value of a variable: `print [var]` or `p [var]`
  - To see the function stack: `where`
  - To re-execute the last command: `<enter>`
  - To quit: `quit`

Example

```
#include <stdio.h>

void foo() {
    printf("This is function foo()\n");
}

int main() {
    int i=0;
    while (i<3) /* No { here! */
        i++;
        foo();
    /* No } here! */
    return 0;
}
```
$ gdb prog1
GNU gdb Red Hat Linux (6.1post-1.20040607.41rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.

(gdb) break foo
Breakpoint 1 at 0x4004ac: file prog1.c, line 4.
(gdb) run
Starting program: /home/gpierre/prog1

Breakpoint 1, foo () at prog1.c:4
  4     printf("This is function foo()\n");
(gdb) where
#0  foo () at prog1.c:4
#1  0x0000000000004004e4 in main () at prog1.c:11
(gdb) up
#1  0x0000000000004004e4 in main () at prog1.c:11
11    foo();

(gdb) list
6
7     int main() {
8     int i=0;
9     while (i<3)
10     i++;
11     foo();
12
13     return 0;
14   }
(gdb) print i
$i1 = 3
(gdb) c
Continuing.
This is function foo()

Program exited normally.
(gdb) quit
$
GDB Can Show More:

```c
struct complex {
    float real;
    float complex;
};

struct mystuct {
    struct complex comp;
    struct mystuct *next;
};

int main() {
    struct mystuct a1 = {{2.3, 1.6}, 0};
    struct mystuct a2 = {{0, -1}, &a1};
    return 0;
}
```

$ gdb prog2
GNU gdb Red Hat Linux (6.1post-1.20040607.41rh)
Copyright 2004 Free Software Foundation, Inc.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.

(gdb) b prog2.c:15
Breakpoint 1 at 0x40049c: file prog2.c, line 15.
(gdb) r
Starting program: /home/gpierre/work/courses/syssprogl5.debug/prog2

Breakpoint 1, main () at prog2.c:15
    15    return 0;
(gdb) p m2
$1 = {comp = {real = 0, complex = -1}, next = 0x7ff003c0}
(gdb) p m2.next
$2 = (struct mystuct *) 0x7ff003c0
(gdb) p *m2.next
$3 = {comp = {real = 2.299999985, complex = 1.600000002}, next = 0x0}
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
Debugging After Core Dump

Did you ever wonder what “core dump” means?
- When a program crashes, your operating system saves the whole state of the program’s memory into a file
- So that you can have a look and identify what went wrong
  - Which instruction caused the crash
  - What was the state of the function stack
  - What was the contents of variables
- It is up to you to figure out why the program reached that state!

Debugging After Core Dump

Programs dump a core:
- Upon a segmentation fault (your program tried to access a protected piece of memory)
- Upon a bus error (your program tried to make a non-aligned memory access)
  - E.g., integer’s memory addresses must be multiples of 4
- When a program calls abort()
- When an assert()ion fails
- Sometimes the system will not dump any core
  - Type this command, then run your program again in the same terminal:

```
ulimit -c unlimited
```
$ cat prog3.c
int main() {
    int *i;
    /* Variable i is not initialized! */
    printf("*i=%d\n",*i);
}
$ ./prog3
Segmentation fault (core dumped)
$ gdb prog3 core.8130
(...)

Core was generated by './prog3'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib64/tls/libc.so.6...done.
Loaded symbols for /lib64/tls/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
#0  0x00000000000400b4 in main () at prog3.c:3
        printf("*i=%d\n",*i);
(gdb) print i
$1 = (int *) 0x0
(gdb) quit
$

DDD: The Data Display Debugger

- When you have complex data structures it can be tedious to explore them with gdb
  - DDD is especially good at displaying them graphically
- DDD is not a debugger but just a graphical interface
  - It starts GDB for you
  - Every action you make is translated into a GDB command
  - It displays the result graphically
- It can also interface to the Java debugger, perl, bash, etc.
- Current version: 3.3.11
Valgrind

- GDB does little to detect memory leaks
  - It merely shows you what is going on
  - It does not "know" what is good or bad programming
  - Memory leaks do not directly produce an error
    ⇒ They are hard to locate with GDB

- Valgrind is specialized in memory-related bugs
  - Current version: 3.0.0
  - http://valgrind.org/

- Valgrind is a set of tools
  - Two memory error detectors, a thread error detector, a cache profiler and a heap profiler.
  - The most important one: Memcheck (memory debugger)

---

Example

```c
void f() {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;  // problem 1: heap block overrun */
    // problem 2: memory leak -- x is not freed */
}

int main() {
    f();
    return 0;
}
```
Splint

• Very long ago, somebody wrote a program called lint
  ▶ It took a C source file as input
  ▶ And checked for common mistakes

• Even better: splint
  ▶ http://www.splint.org/
  ▶ It checks for common bugs
  ▶ Focuses mostly on security holes (but not only)

• splint will issue warnings
  ▶ Some warnings you may decide to ignore (at your own risk)
  ▶ Remember: even if splint does not display anything, this does not
mean that your program is correct!
Example

- Let us write a very bad program:

```c
#include <stdio.h>

int main() {
    char buf[128];
    gets(buf);
    printf(buf);
    return 0;
}
```

$ splint foo.c
Splint 3.1.1 --- 15 Jun 2004

foo.c: (in function main)
foo.c:5:3: Use of gets leads to a buffer overflow vulnerability. Use fgets instead: gets
    Use of function that may lead to buffer overflow. (Use -bufferoverflowhigh to inhibit warning)
foo.c:5:3: Return value (type char *) ignored: gets(buf)
    Result returned by function call is not used. If this is intended, can cast result to (void *) to eliminate message. (Use -retvalother to inhibit warning)
foo.c:6:3: Format string parameter to printf is not a compile-time constant: buf
    Format parameter is not known at compile-time. This can lead to security vulnerabilities because the arguments cannot be type checked. (Use -formatconst to inhibit warning)

Finished checking --- 3 code warnings
$
Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), and J.Kim (KAIST).