# C Programming and Debugging

Week 5

# C Programming

# Types

- Subset of C++
- Built-in types
  - Integers, Floating-point, Character strings
  - No bool
- Compiling 'C' only
  - gcc -std=c99 binsortu.c
- No classes, but we have structs
- No methods and access modifiers in C structs

```
struct Song
{
    char title[64];
    char artist[32];
    char composer[32];
    short duration; // Playing time in seconds.
    struct Date published; // Date of publication.

};
```

# Declaring Pointers

```
int *iPtr;   //Declare iPtr as a pointer to int


iPtr = &iVar //Let iPtr point to the variable iVar


int iVar = 77; // Define an int variable
int *iPtr = &iVar; // Define a pointer to to it
```

# Dereferencing Pointers

```
double x, y, *ptr;          // Two double variables
                            and a pointer to double.

ptr = &x;                   // Let ptr point to x.

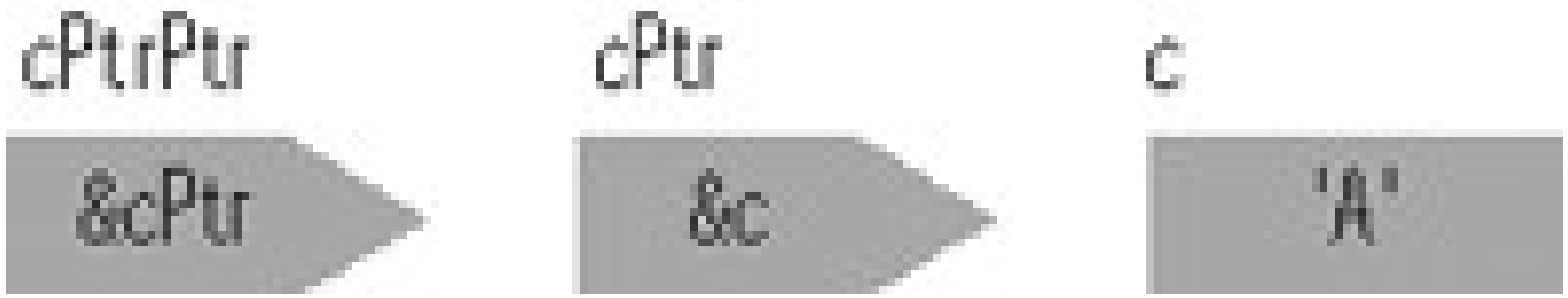
*ptr = 7.8;                 // Assign the value 7.8 to
                            the variable x.

*ptr *= 2.5;                // Multiply x by 2.5.

y = *ptr + 0.5;             // Assign y the result of
                            the addition x + 0.5.
```

# Pointer to Pointers

```
char c = 'A', *cPtr = &c, **cPtrPtr = &cPtr
```



# Pointers to Functions

```
double (*funcPtr)(double, double);
double result;
funcPtr = pow; // Let funcPtr point to the function pow( ).
               // The expression *funcPtr now yields the
               // function pow( ).
result = (*funcPtr)( 1.5, 2.0 );
               // Call the function referenced by funcPtr.
result = funcPtr( 1.5, 2.0 ); // The same function call.
```

# typedef Declarations

- Easy way to use types with complex names

```
typedef struct Point { double x, y; } Point_t;

typedef struct
{
   Point_t top_left;
   Point_t bottom_right;
} Rectangle_t;
```

# Dynamic Memory Management

- malloc(size_t size): allocates a block of memory whose size is at least *size.*
- free(void *ptr): frees the block of memory pointed to by ptr

```
Rectangle_t *ptr = (Rectangle_t*) malloc(sizeof(Rectangle_t));

if(ptr == NULL){
  printf("Something went wrong in malloc");
  exit(-1);
}
else {
  //Perform tasks with the memory
  //
  //
  free(ptr);
  ptr = NULL;
}
```

- realloc(void *ptr, size_t newSize) : Resizes allocated memory
  - `ptr = (Rectangle_t*) realloc(ptr, 3 * sizeof(Rectangle_t));`

# Opening & Closing Files

FILE *fopen( const char * restrict *filename*, const char * restrict *mode* );

int fclose( FILE *fp );

```
FILE *fPtr = NULL;
fPtr = fopen("data.txt", "w"); //r - read, w - write, a - append
if(fPtr != NULL)
{
  fputs("Sample code", fPtr);
  fclose(fPtr);
}
else
{
  printf("Something went wrong in fopen");
  exit(-1);
}
```

# Reading Characters

- Common streams and their file pointers
  - Standard input - stdin
  - Standard output - stdout
  - Standard error - stderr
- Reading/Writing characters
  - getc( FILE *_fp_ );
  - putc(int c, FILE *pf);
- Reading/Writing Lines
  - char *fgets( char *_buf_, int _n_, FILE *_fp_ );
  - int fputs( const char *_s_, FILE *_fp_ );

# Formatted Input/Output

- Formatted Output
  - int fprintf( FILE * restrict *fp*, const char * restrict *format*, ... );
  - int fscanf( FILE * restrict *fp*, const char * restrict *format*, ... );
- The format string
  - ```
    int score = 120;
    char player[ ] = "Mary";
    printf( "%s has %d points.\n", player, score );

    fprintf(stdout, "%s has %d points.\n", player, score );
    fscanf(stdin, "%d", &score);

    fprintf(fPtr, "%s has %d points.\n", player, score );
    fscanf(fPtr, "%d", &score);
    ```
- Format Specifiers
  - %s – strings
  - %d – decimal integers
  - %f – floating point
    ```

# GDB - Debugging

# Process Layout

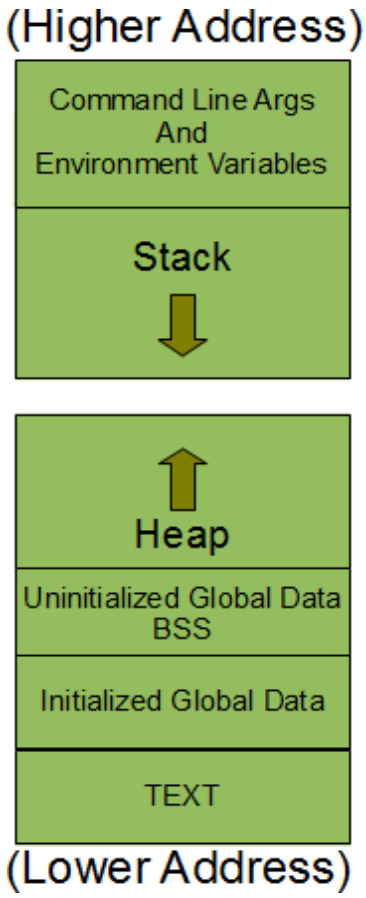



Image source : thegeekstuff.com

- TEXT segment
  - Contains machine instructions to be executed
- Global Variables
  - Initialized
  - Uninitialized
- Heap segment
  - Dynamic memory allocation
  - malloc, free
- Stack segment
  - Push frame: Function invoked
  - Pop frame: Function returned
  - Stores
    - Local variables
    - Return address, registers, etc
- Command Line arguments and Environment Variables

# Call Stack

```
#include<stdio.h>
void function1(int arg);
void function2(int arg);

int main(int argc, char** argv)
{
    printf("Main function started\n");
    int fArgs = 10;
    function1(fArgs);
    return 0;
}

void function1(int arg)
{
    printf("Function1 arg is %d\n", arg);
    arg = arg - 1;
    function2(arg);
    return;
}

void function2(int arg)
{
    printf("Function2 arg is %d\n", arg);
    arg = arg + 1;
    return;
}
```

# Compiling

- `gcc –o ExecutableFile -g main.c`
- The –o option indicates the name of the binary/program to be generated
- The –g option indicates to include symbol and source-line info for debugging

# Displaying Source Code

- list *filename*: *line_number*
  - Displays source code centered around the line with the specified line number.
- list *line_number*
  - If you do not specify a source filename, then the lines displayed are those in the current source file.
- list *from*,[ *to*]
  - Displays the specified range of the source code. The *from* and *to* arguments can be either line numbers or function names. If you do not specify a *to* argument, list displays the default number of lines beginning at *from*.
- list *function_name*
  - Displays source code centered around the line in which the specified function begins.
- list, l
  - The list command with no arguments displays more lines of source code following those presented by the last list command. If another command executed since the last list command also displayed a line of source code, then the new list command displays lines centered around the line displayed by that command.

# Breakpoints

- break [ *filename*:] *line_number*
  - Sets a breakpoint at the specified line in the current source file, or in the source file *filename*, if specified.
- break *function*
  - Sets a breakpoint at the first line of the specified function.
- break
  - Sets a breakpoint at the next statement to be executed. In other words, the program flow will be automatically interrupted the next time it reaches the point where it is now.

# Deleting, Disabling, and Ignoring BP

- delete [ *bp_number* | *range*]
- d [ *bp_number* | *range*]
  - Deletes the specified breakpoint or range of breakpoints. A delete command with no argument deletes all the breakpoints that have been defined. GDB prompts you for confirmation before carrying out such a sweeping command:
  - (gdb) **d** Delete all breakpoints? (y or n)
- disable [ *bp_number* | *range*]
  - Temporarily deactivates a breakpoint or a range of breakpoints. If you don't specify any argument, this command affects all breakpoints. It is often more practical to disable breakpoints temporarily than to delete them. GDB retains the information about the positions and conditions of disabled breakpoints so that you can easily reactivate them.
- enable [ *bp_number* | *range*]
  - Restores disabled breakpoints. If you don't specify any argument, this command affects all disabled breakpoints.
- ignore *bp_number iterations*
  - Instructs GDB to pass over a breakpoint without stopping a certain number of times. The ignore command takes two arguments: the number of a breakpoint, and the number of times you want it to be passed over.

# Conditional Breakpoints

- break [*position*] if *expression*

```
(gdb) s
27 for ( i = 1; i <= limit ; ++i )
(gdb) break 28 if i == limit - 1

Breakpoint 1 at 0x4010e7: file gdb_test.c, line 28.
```

# Resuming Execution After a Break

- continue [ *passes*] , c [ *passes*]
  - Allows the program to run until it reaches another breakpoint, or until it exits if it doesn't encounter any further breakpoints. The *passes* argument is a number that indicates how many times you want to allow the program to run past the present breakpoint before GDB stops it again. This is especially useful if the program is currently stopped at a breakpoint within a loop. See also the ignore command, described in the previous section "Working with Breakpoints."
- step [ *lines*] , s [ *lines*]
  - Executes the current line of the program, and stops the program again before executing the line that follows. The step command accepts an optional argument, which is a positive number of source code lines to be executed before GDB interrupts the program again. However, GDB stops the program earlier if it encounters a breakpoint before executing the specified number of lines. If any line executed contains a function call, step proceeds to the first line of the function body, provided that the function has been compiled with the necessary symbol and line number information for debugging.
- next [ *lines*] , n [ *lines*]
  - Works the same way as step, except that next executes function calls without stopping before the function returns, even if the necessary debugging information is present to step through the function.
- quit, q
  - Quit debugger
- finish
  - To resume execution until the current function returns, use the finish command. The finish command allows program execution to continue through the body of the current function, and stops it again immediately after the program flow returns to the function's caller. At that point, GDB displays the function's return value in addition to the line containing the next statement.

# Analyzing the Stack

- Bt
  - Shows the call trace
- info frame
  - Displays information about the current stack frame, including its return address and saved register values.
- info locals
  - Lists the local variables of the function corresponding to the stack frame, with their current values.
- info args
  - List the argument values of the corresponding function call.

# Displaying Data

- p [/*format*] [*expression*]
- Output Formats
  - d: Decimal notation. This is the default format for integer expressions.
  - u: Decimal notation. The value is interpreted as an unsigned integer type.
  - x: Hexadecimal notation.
  - o: Octal notation.
  - t: Binary notation. Do not confuse this with the x command's option b for "byte," described in the next subsection.
  - c: Character, displayed together with the character code in decimal notation.

# Watchpoints

- watch *expression*
  - The debugger stops the program when the value of *expression* changes.
- rwatch *expression*
  - The debugger stops the program whenever the program reads the value of any object involved in the evaluation of *expression*.
- awatch *expression*
  - The debugger stops the program whenever the program reads or modifies the value of any object involved in the evaluation of *expression*.

# Debug flags

- Compile it:
  - Compiled with the -g flag to add debugging information into the executable
  - gcc –o gdb_example –g gdb_example.c
- Run:
  - ./gdb_example
- Debug
  - gdb gdb_example
- Debug ls
  - gdb --args ./ls -lt /tmp/wwi-armistice /tmp/now /tmp/now1

# Sample Debugging Session

- Do a list (two times)
  - l
  - l
- Set a breakpoint at line 14, where we call the swap function
  - b 14
- Step into the swap function
  - S
- Print the contents of *p1 and *p2 out
  - p *p1
  - p *p2
- Execute the next three lines
  - n
  - n
  - n
- Print the contents of *p1 and *p2 out
  - p *p1
  - p *p2
- Print the contents of p1 and p2 out
  - p *p1
  - p *p2
- Continue the program
  - c
- Quit
  - q

# Laboratory

# Getting Started

- Download the buggy version of coreutils
- configure, make
- Test the bug

```
$ touch -d '1918-11-11 11:00 GMT' /tmp/wwi-armistice
$ touch /tmp/now
$ sleep 1
$ touch /tmp/now1
$ ls -lt /tmp/wwi-armistice /tmp/now /tmp/now1
```

- Use gdb to figure out what's wrong (running from the directory where the compiled ls lives)

```
$ gdb --args ./ls –lt /tmp/wwi-armistice /tmp/now tmp/now1
```

# Getting Started

- Unix Epoch time
  - Timestamps stored as seconds or nanoseconds elapsed since 1970 Jan 1st 00:00 (UNIX Epoch)
  - ctime – File status changes
  - mtime – File contents modified
  - atime – File access
- Qsort

```
void qsort (void* base, //Records to be sorted
            size_t num, //Number of records
            size_t size, //Size of each record
            int (*compare)(const void*,const void*)  //”Compare” function that specifies ordering
            );
```

- Compare
  - `int (*comp)(const void* r1, const void* r2) // Takes 2 records as arguments and returns an int`
  - Return value < 0 then 'r1' goes before 'r2'
  - Return value = 0 then 'r1' and 'r2' are equivalent
  - Return value > 0 then 'r2' goes before 'r1'

# Construct a Patch

- Construct a patch containing your fix in a patch file called lab5.patch
- It should contain a ChangeLog entry followed by the output of diff -u

# Homework

- Lexicographic byte comparison to compare each record, in style of **memcmp**
- **stdio.h** for I/O
- **malloc, realloc, free** to allocate storage
- **qsort** to sort the data
- **strtoul** to convert a string to a long
- Errors should be reported to **stderr** and should cause the program to **exit** with status 1