# C Programming

and

# Debugging

CS 35L

Spring 2020 – Section 7

# Types

- Subset of C++ (very similar)
- Built-in types:
  - Integers, Floating-point, character strings
  - No bool, false is 0 and true is anything else
- Compiling ‘C’ only
  - gcc -std=c99 binsortu.c

# Types

- 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;
    struct Date published;
};
```

# Declaring Pointers

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

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

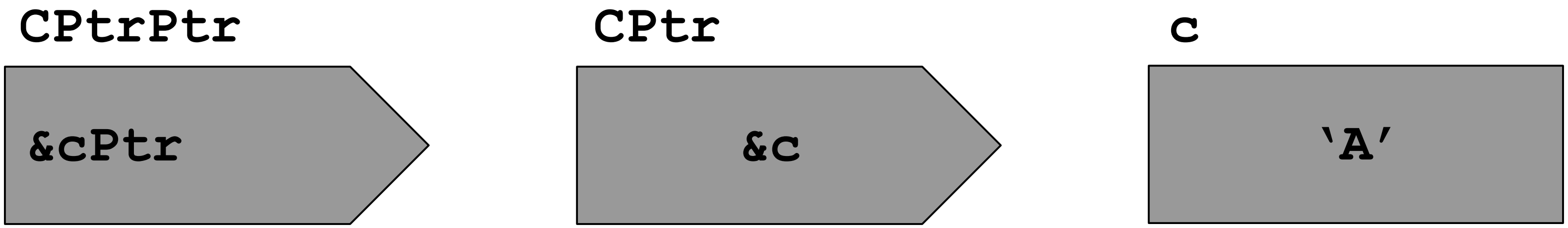
```
int iVar = 77;     // Define an int variable
int *iPtr = &iVar; // Define a pointer to to it
```

# 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’;
char *cPtr = &c;
char **cPtrPtr = &cPtr;
```



CPtrPtr

&cPtr

CPtr

&c

c

‘A’

# Pointers to Functions

```
double (*funcPtr)(double, double);
double result;

// Let funcPtr point to the function pow( ).
// The expression *funcPtr now yields the
// function pow( ).
funcPtr = pow;

// Call the function referenced by funcPtr.
result = (*funcPtr)( 1.5, 2.0 );

// The same function call.
result = funcPtr( 1.5, 2.0 );
```

# 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;

typedef double banana;
banana yellow = 32.0;
```

# Dynamic Memory Management

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

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

if(ptr == NULL) {
    printf(“Malloc failed!”);
    exit(-1);
}
else {
    //Perform tasks with the memory
    free(ptr);
    ptr = NULL;
}


ptr = (Rectangle_t*)
    realloc(ptr, 3*sizeof(Rectangle_t))
```

# Opening & Closing Files

```
FILE *fptr;

FILE *fopen( const char * restrict
  filename, const char * restrict mode
  );
int fclose( FILE *fp );
```

Common Streams and their file pointers

- Standard input: `stdin`
- Standard output: `stdout`
- Standard error: `stderr`

# Reading Characters

- Reading/Writing characters
  - `getc( FILE *fp );`
  - `putc(int c, FILE *pf);`
- Reading/Writing Lines
  - `char* fgets(`<br>
    `char *buf, int n, FILE *fp);`
  - `int fputs(`<br>
    `const char *s, FILE *fp);`

# FormattedOutput

- Formatted Output to stdout
  - `int printf(const char * restrict format, ... );`
- The format string

```
double score = 42.5;
int player_number = 26;
char player[ ] = "Mary";

printf("%s (#%d) has %f points.\n",
         player, player_number, score);

// Outputs: Mary (#26) has 42.5 points.
```

- Format Specifiers
  - %s → Strings
  - %d → Decimal integers
  - %f → floating points

# Formatted Input/Output

- Formatted Input/Output For Files (including stdin/stdout)
  - 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";
FILE* fPtr = fopen("blah", "rw");

fprintf(stdout, "%s has %d points\n", player, score);
fprintf(fPtr, "%s has %d points.\n", player, score);
// Outputs: Mary has 120 points.

fscanf(stdin, “%d”, &score);
fscanf(fPtr, “%d”, &score);
// Read in a single integer number and write it into score
```

# Ternary Operator `?:`

- Short form for a conditional assignment:

```
result = a > b ? x : y;
```

- Equivalent to:

```
if(a > b) {
  result = x;
}
else {
  result = y;
}
```

# Sample Program

```
#include <stdio.h>

// Method definition before use
  void printHelloWorld();

int main(char[] argv) {
  printHelloWorld();
  return 0;
}

void printHelloWorld() {
  printf(“%s\n”, “Hello World!”);
}
```

# Compiling

- `gcc –o FooBarBinary -g foobar.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
- For more info, `man gcc`

# GDB - Debugging

# Debugging Process

- Notice a bug - “Huh, that’s weird.”
- Reproduce the bug - “Well, that’s bad.”
- Simplify program input - “Is it that simple?”
- Try (and probably fail) to find bug by eye
  - “Where the $%^# is it?”
- Use a debugger to isolate problem
  - “Aha! That’s why . . .”
- Fix the problem - “And now it’s fixed.”

# Debugger

- A program that is used to run and debug other (target) programs
- Advantages:

  Programmer can:

  – step through source code line by line
    - each line is executed on demand

  – interact with and inspect program at run-time

  – If program crashes, the debugger outputs where and what happened when it crashed

# GDB – GNU Debugger

- Debugger for several languages
  - C, C++, Java, Objective-C… more
- Allows you to inspect what the program is doing at a certain point during execution
- Logical errors and segmentation faults are easier to find with the help of gdb

# Process Layout

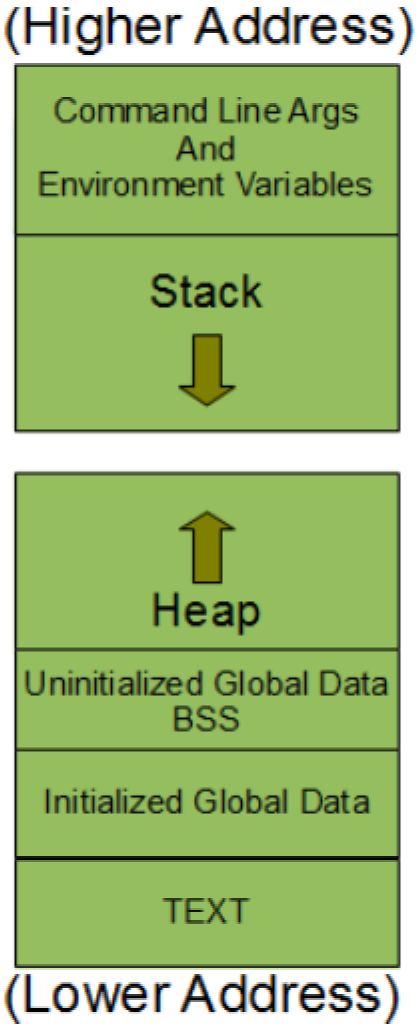


Image source : thegeekstuff.com

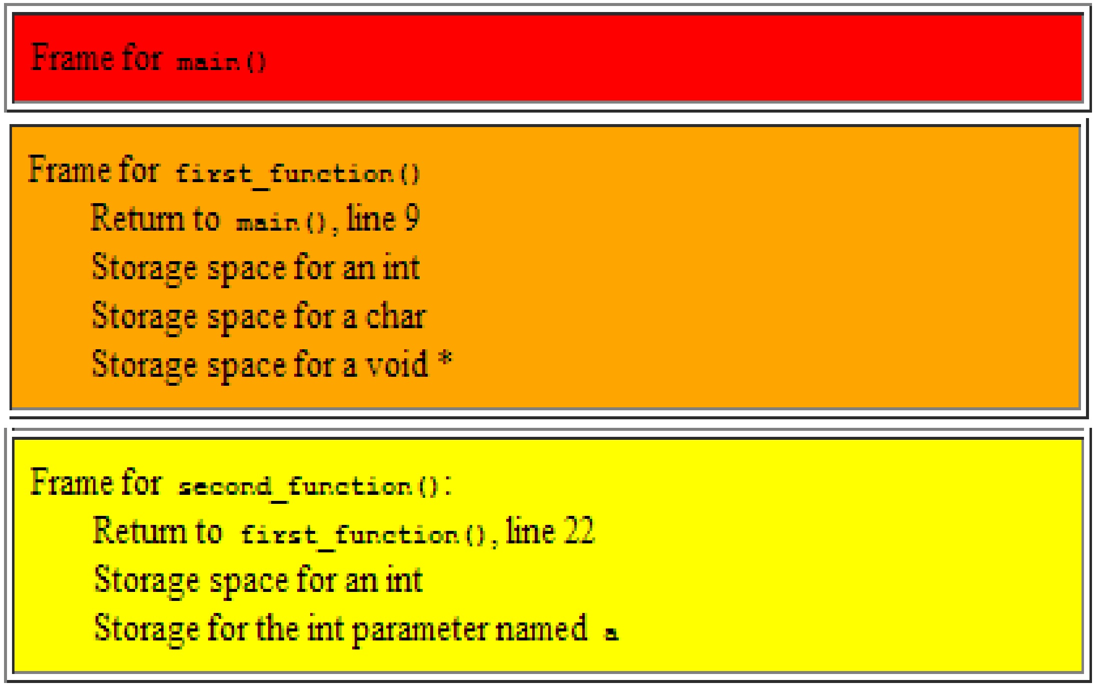
- TEXT segment
  - Contains machine instructions to be executed
- Global Variables
  - Initialized
  - Uninitialized
- Heap segment
  - Dynamic memory allocation
  - malloc, free
- Stack segment
  - Push frame: Function invoked
  - Pop frame: Function returned
  - Stores
    - Local variables
    - Return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

# Stack Info

- A program is made up of one or more functions which interact by calling each other
- Every time a function is called, an area of memory is set aside for it. This area of memory is called a **stack frame** and holds the following crucial info:
  - storage space for all the local variables
  - the memory address to return to when the called function returns
  - the arguments, or parameters, of the called function
- Each function call gets its own stack frame. Collectively, all the stack frames make up the **call stack**

# Stack Frames and the Stack

```
1    #include <stdio.h>
2    void first_function(void);
3    void second_function(int);
4
5    int main(void)
6    {
7       printf("hello world\n");
8       first_function();
9       printf("goodbye goodbye\n");
10
11      return 0;
12   }
13
14
15   void first_function(void)
16   {
17      int imidate = 3;
18      char broiled = 'c';
19      void *where_prohibited = NULL;
20
21      second_function(imidate);
22      imidate = 10;
23   }
24
25
26   void second_function(int a)
27   {
28      int b = a;
29   }
```



Frame for `main()`

Frame for `first_function()`
- Return to `main()`, line 9
- Storage space for an int
- Storage space for a char
- Storage space for a void *

Frame for `second_function()`:
- Return to `first_function()`, line 22
- Storage space for an int
- Storage for the int parameter named `a`

When main() returns, the program ends

[illegible]

execution within second_function()

# 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.
- 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*.

# Using GDB

1. **Compile Program**
   - Normally: `$ gcc [flags] <source files> -o <output file>`
   - Debugging: `$ gcc [other flags]` **`-g`** `<source files> -o <output file>`
     - enables built-in debugging support

2. **Specify Program to Debug**
   - `$ gdb <executable>`

     or

   - `$ gdb`
   - `(gdb) file <executable>`

# Using GDB

3. **Run Program**
   - `(gdb) run` or
   - `(gdb) run [arguments]`
4. **In GDB Interactive Shell**
   - Tab to Autocomplete, up-down arrows to recall history
   - `help [command]` to get more info about a command
5. **Exit the gdb Debugger**
   - `(gdb) quit`

# *Sample Debugging Session*

```c
#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;
}
```

# Gdb pointers

- Gdb cheat sheet
- Gdb command tutorial and slides
- Running gdb with emacs