ECE264: Advanced C Programming

Summer 2019

Week 1: Tools, Program Layout, Data Types and Structs

Git

- Version Control System
  - Manage versions of your code – access to different versions when needed
  - Lets you collaborate
- 'Repository' – virtual storage
  - Local and Remote Repository
    - Local is working copy

Git – Initializing Repositories

- Getting started with local working copies:
  - git init

```
Terminal
[ecegrid-thin4:~/ECE264/dem0] hegden$ls -a
.  ..
[ecegrid-thin4:~/ECE264/dem0] hegden$git init
Initialized empty Git repository in /home/min/a/hegden/ECE264/dem0/.git/
[ecegrid-thin4:~/ECE264/dem0] hegden$ls -a
.  ..  .git
[ecegrid-thin4:~/ECE264/dem0] hegden$
```

  - git clone (when a remote repository on github.com exists)

```
Terminal
[ecegrid-thin4:~/ECE264] hegden$ls -a
.  ..
[ecegrid-thin4:~/ECE264] hegden$git clone git@github.com:ece264summer2019/dem0.git
Cloning into 'dem0'...
warning: You appear to have cloned an empty repository.
[ecegrid-thin4:~/ECE264] hegden$ls -a
.  ..  dem0
[ecegrid-thin4:~/ECE264] hegden$cd dem0/
[ecegrid-thin4:~/ECE264/dem0] hegden$ls -a
.  ..  .git
[ecegrid-thin4:~/ECE264/dem0] hegden$
```

Git – Adding Content

- Staging

```
Terminal
[ecegrid-thin4:~/ECE264/dem0] hegden$echo "This repository is created for demo purposes." > README.txt
[ecegrid-thin4:~/ECE264/dem0] hegden$git add README.txt
```

- Commit (save changes in local repository)

```
[ecegrid-thin4:~/ECE264/dem0] hegden$git commit -m "My first commit"
[master ab680c6] My first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
```

- Save changes in remote repository (guard against accidental deletes)

```
[ecegrid-thin4:~/ECE264/dem0] hegden$git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Writing objects: 100% (3/3), 290 bytes | 145.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:ece264summer2019/dem0.git
   3dccc4f..ab680c6  master -> master
```

Git – Releasing Code

- Tagging
  - make sure there are no unsaved changes in local repository

```
[ecegrid-thin4:~/ECE264/dem0] hegden$git status .
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

```
[ecegrid-thin4:~/ECE264/dem0] hegden$git tag -a RELEASE_V0.1 -m "First release"
```

- Save tags in remote repository

```
[ecegrid-thin4:~/ECE264/dem0] hegden$git push --tags
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 176 bytes | 176.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To github.com:ece264summer2019/dem0.git
 * [new tag]         RELEASE_V0.1 -> RELEASE_V0.1
```

Git – Recap..

- Please read https://git-scm.com/book/en/v2 for details

```
1.  git clone (creating a local working copy)
2.  git add (staging the modified local copy)
3.  git commit (saving local working copy)
4.  git push (saving to remote repository)
5.  git tag (Naming the release with a label)
6.  git push --tags (saving the label to remote)
```

Makefile

- Is a file, contains instructions for the ‘make’ program to generate a target (executable).
- Generating a target involves:
    1. Preprocessing (e.g. strips comments, conditional compilation etc.)
    2. Compiling (.c -> .s files, .s -> .o files)
    3. Linking (e.g. making printf available)
- A Makefile typically contains directives on how to do steps 1, 2, and 3.

Makefile - Format

- Contains series of ‘rules’-

```
target: dependencies
[TAB] system command(s)
```

Note that it is important that there be a TAB character before the command (not spaces).

Example,

```
testgen: testgen.c
	gcc testgen.c –o testgen
```

- And Macro/Variable definitions -

```
CFLAGS = -std=c99 -g -Wall -Wshadow --pedantic -Wvla –Werror

GCC = gcc
```

Makefile - Usage

- The ‘make’ command (Assumes that a file by name ‘makefile’ or ‘Makefile’. exists)

```
[ecegrid-thin4:~/ECE264/dem0] hegden$cat makefile
GCC=gcc
CFLAGS=-std=c99 -g -Wall -Wshadow --pedantic -Wvla -Werror
testgen: testgen.c
	$(GCC) $(CFLAGS) testgen.c -o testgen
clean:
	rm testgen
[ecegrid-thin4:~/ECE264/dem0] hegden$make
gcc -std=c99 -g -Wall -Wshadow --pedantic -Wvla -Werror testgen.c -o testgen
[ecegrid-thin4:~/ECE264/dem0] hegden$
```

- To know more, please read: https://www.gnu.org/software/make/manual/html_node/index.html#Top

Sorting

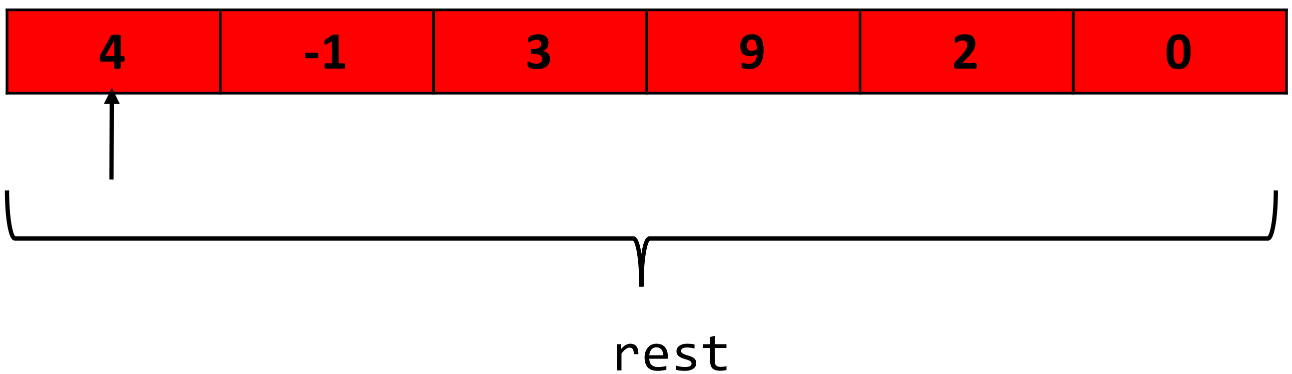
- Arranging the elements of a list in a particular order.
- E.g. sorting list of names in lexicographical order, sorting numerical input in ascending order, etc.
- Used often as a pre-processing step in optimizing computation.
  - Easier and faster to locate items

Sorting - Selection sort

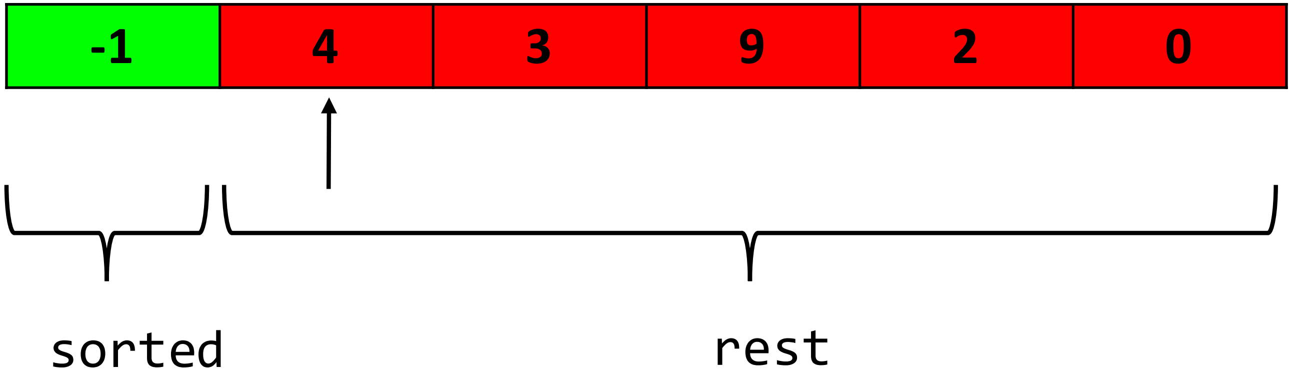
- Repeatedly find the minimum element in the unsorted array and put it at the beginning.
  - Divides the input array into 2 pieces - `sorted` and `rest`.
  - *All elements* in `sorted` are smaller than *any element* in the `rest` – *invariant*
  - Works by growing `sorted` and shrinking `rest`

Selection sort - example

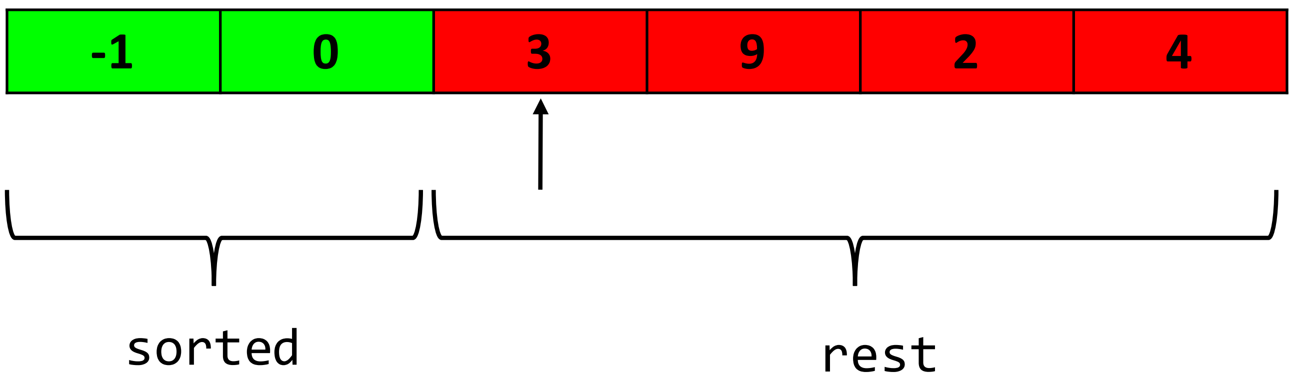
- A cursor dividing `sorted` and `rest`



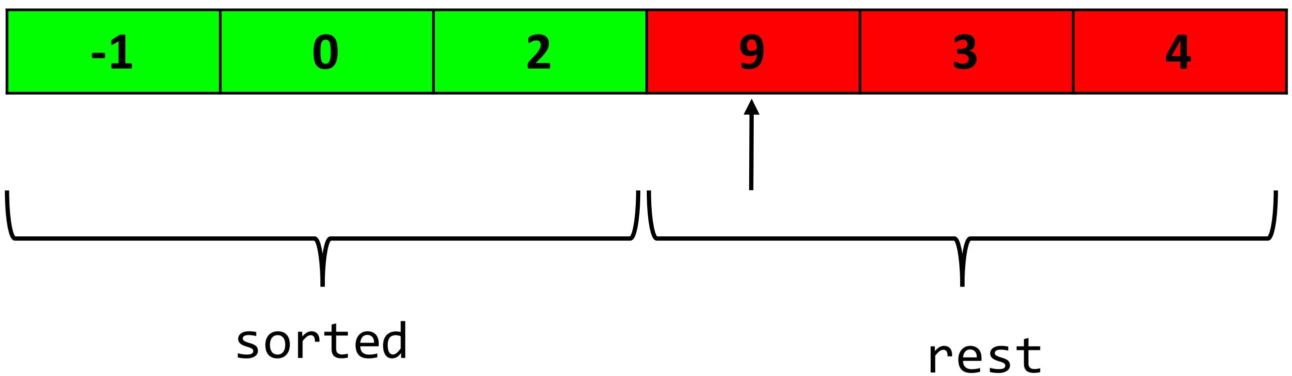
Selection sort - example



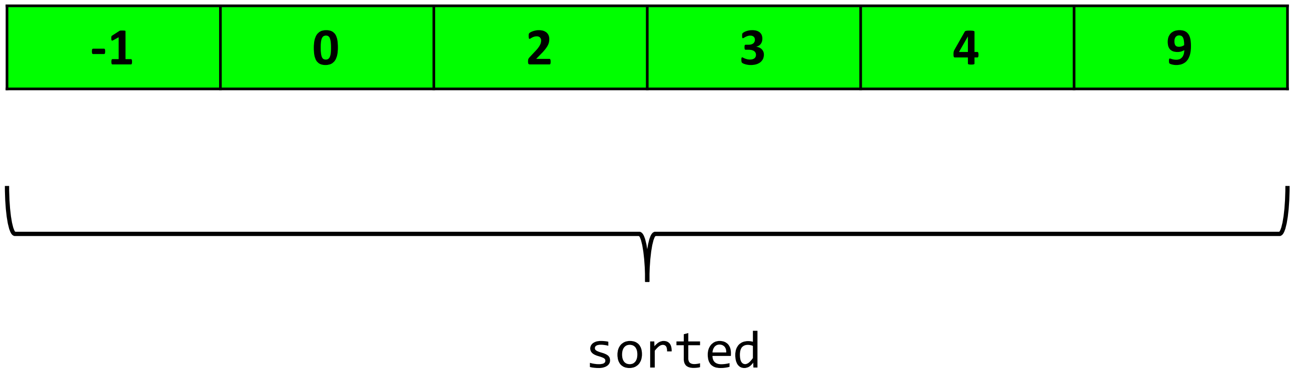
Selection sort - example



Selection sort - example



Selection sort - example



Sorting algorithms - Evaluation

- Many metrics used for evaluating sorting algorithms.
- Two most common metrics are:
    - How many comparisons are involved?
    - How much data movement is involved?

Selection sort - pseudocode

```
int input[N] = //input
int cursor = 0 //initial position of the cursor
for(cursor = 0; cursor < N; cursor++)
 //sorted list from [0,cursor)
 //rest of the list from [cursor, N)
 for(i = cursor; i < N; i++)
   //search the rest of the list to find the smallest value
 //swap the smallest value with the value at input[cursor]
```

Selection sort - Analysis

```
1 int input[N] = //input
2 int cursor = 0 //initial position of the cursor
3 for(cursor = 0; cursor < N; cursor++)
4   //sorted list from [0,cursor)
5   //rest of the list from [cursor, N)
6   for(i = cursor; i < N; i++)
7     //search the rest of the list to find the smallest value
8   //swap the smallest value with the value at input[cursor]
```

- Outer loop (line 3) is moving the cursor, inner loop (line 6) is finding minimum.

How many times does inner loop execute?

Selection sort - Analysis

```
int input[N] = //input
int cursor = 0 //initial position of the cursor
for(cursor = 0; cursor < N; cursor++)
  //sorted list from [0,cursor)
  //rest of the list from [cursor, N)
  for(i = cursor; i < N; i++)
    //search the rest of the list to find the smallest value
  //swap the smallest value with the value at input[cursor]
```

- inner loop runs N times, (N - cursor) iterations every time.

$$= \sum_{i=0}^{N-1} N - i$$

$$= \sum_{i=1}^{N} i \qquad = \frac{N\,(N+1)}{2}$$

Selection sort - Analysis

- outer loop runs for N iterations
- inner loop runs for ~ N(N+1)/2 iterations
  - inner loop dominates

1. Approximately how many array write operations occur?

2. Double the input, how long does Selection sort take?

Number Bases

- We use decimal (base-10), Computers use binary (base-2).
- Binary is difficult to read. So, we use Hexadecimal (base-16).
- Octal (base-8) is the other popular number format.

Number Bases - Hexadecimal

- Hexadecimal uses 16 digits: 0 to 9 and A to F. A to F represent decimal numbers 10 to 15.
- A digit in hexadecimal needs 4 bits. Therefore, a byte of information (8 bits) represents two digits.
- Example:

Decimal	Binary	Hexadecimal
10	1010	0xA
16	1 0000	0x10
43981	1010 1011 1100 1101	0xABCD

How are Numbers Stored in Memory? - Endianness

- Assume an integer needs 4 bytes of storage
  - E.g. 1193 in Hexadecimal = 0x4A9 = 0x 00 00 04 A9 when stored in 4 bytes of memory.
  - How are those 4 bytes ordered in memory? – Endianness
- Two popular formats: Big-Endian and Little-Endian

Big-Endian

- Most-significant-byte (MSB) at low-address and least-significant-byte (LSB) at high-address
  - E.g. 1193 = **0x00 00 04 A9** (= 4 * 16^2 + A * 16 + 9)
  - MSB (0x00) is written at lower address, LSB (0xA9) is written at higher address.

	0000 0000 **(00)**	0000 0000 **(00)**	0000 0100 **(04)**	1010 1001 **(A9)**
Address:	0x00000001	0x00000002	0x00000003	0x00000004

  - Motorola 68000 Series, IBM-Z Mainframes.

Little-Endian

- Most-significant-byte (MSB) at high-address and least-significant-byte (LSB) at low-address
  - E.g. 1193 = **0x00 00 04 A9** (= 4 * 16^2 + A * 16 + 9)
  - MSB (0x00) is written at higher address, LSB (0xA9) is written at lower address.

	1010 1001 **(A9)**	0000 0100 **(04)**	0000 0000 **(00)**	0000 0000 **(00)**
Address:	0x00000001	0x00000002	0x00000003	0x00000004

  - Intel x86 Architecture

Little-Endian

- What gets flipped in Little-endian?

Flipped	Not-Flipped
• Bytes	• Bits within a byte, Hex-digits within a byte
• Multi-byte numbers (e.g. int, long, float) and addresses	• Array elements and Struct fields

Endianness

- Fortunately, we don’t have to worry about endianness.
  - You don’t have to reverse bytes when you read an integer.
  - Compiler and the processor do the job for you.
- However, you need to be aware of endianness when inspecting memory contents.
  - E.g. when using GDB while debugging.

Program Layout in Memory

- Why know it?
  - Debug programs
  - Design software for constrained devices (e.g. embedded systems)
  - Design robust (secure) software

Program Layout in Memory

- A program's memory space is divided into four segments:

1. Text
   - source code of the program
2. Data
   - Broken into uninitialized and initialized segments; contains space for global and static variables. E.g. `int x = 7; int y;`
3. Heap
   - Memory allocated using malloc/calloc/realloc
4. Stack
   - Function arguments, return values, local variables, special registers.

Detour - Stacks

Real Stack

Hardware Stack

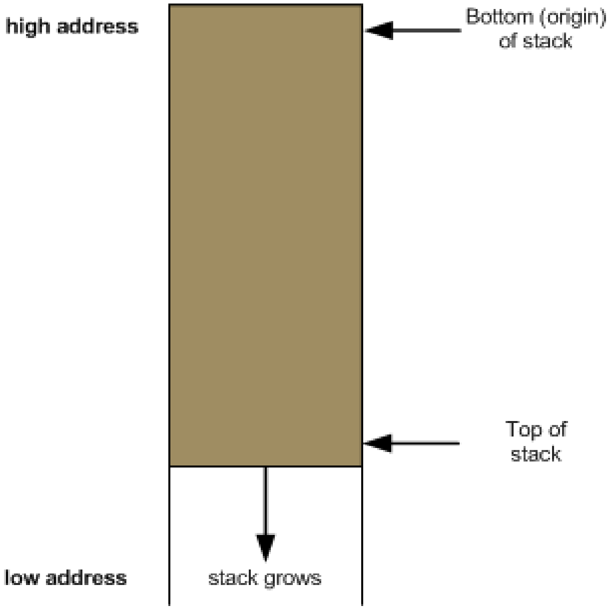
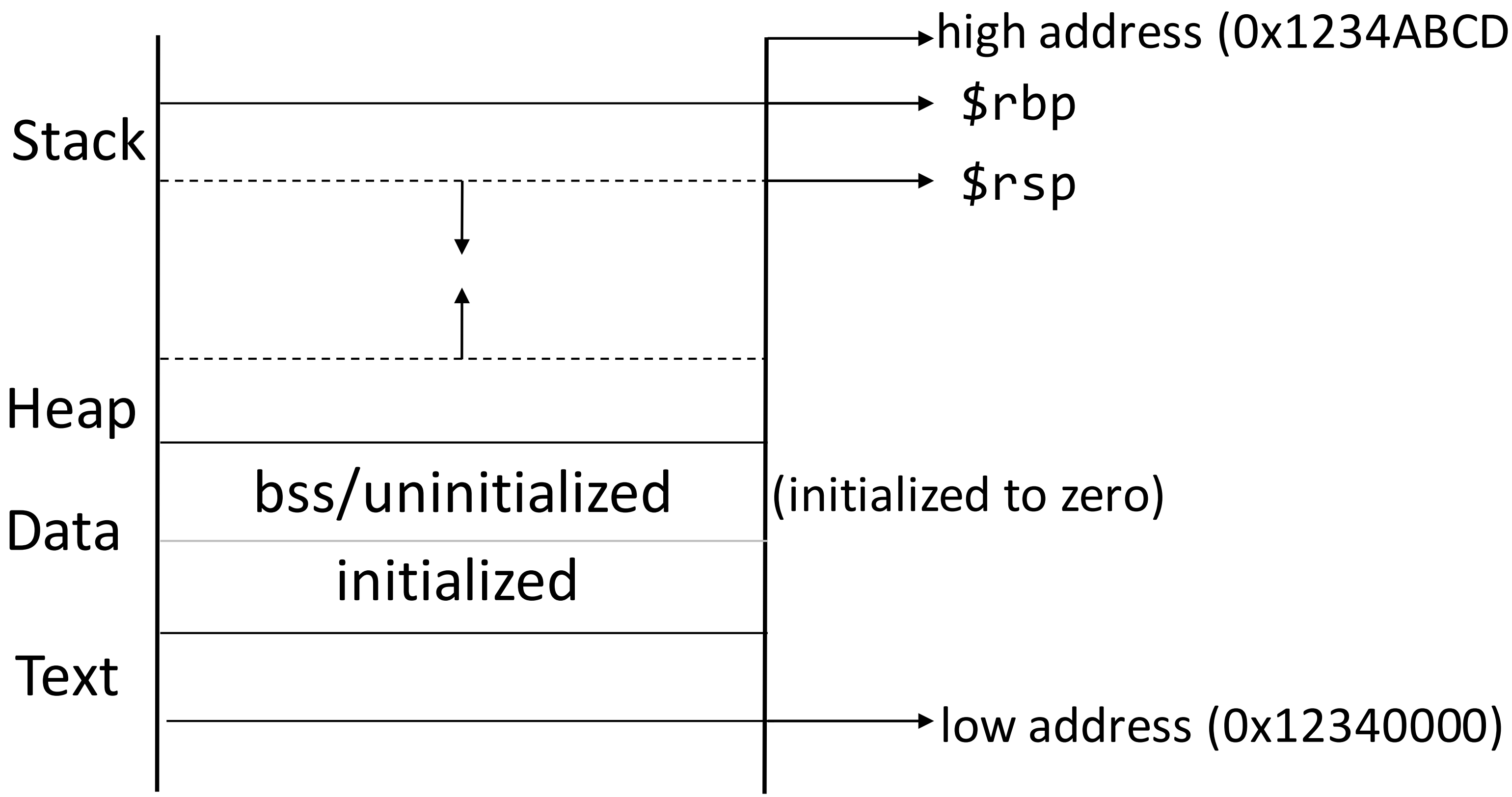


Image source: https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/

Stack Frame

- A sub-segment of memory on the stack space
  - Special registers `$rbp` and `$rsp` track the bottom and top of the stack frame.
  - Example: when `main` calls function `foo`
    1. The following are pushed on to stack:
       - `foo`'s arguments
       - Space for `foo`'s return value
       - Address of the next instruction executed (in `main`) when `foo` returns
       - Current value of `$rbp`
    2. `$rsp` is automatically updated (decremented) to point to current top of the stack.
    3. `$rbp` is assigned the value of `$rsp`

Program Layout in Memory



Question ?

Where are the command-line arguments stored?

GDB

- GNU Debugger – A tool for inspecting your C programs
  - How to begin inspecting a program using gdb?
  - How to control the execution?
  - Misc – displaying stack frames, visualizing assembler code.
  - How to display, interpret, and alter memory contents of a program using gdb?

GDB

- Compile your programs with –g option

```
[ecegrid-thin4:~/ECE264] hegden$gcc gdbdemo.c -o gdbdemo -g
[ecegrid-thin4:~/ECE264] hegden$
```

```c
#include<stdio.h>
int foo(int a, int b)
{
    int x = a + 1;
    int y = b + 2;
    int sum = x + y;

    return x * y + sum;
}

int main()
{
   int ret = foo(10, 20);
   printf("value returned from foo: %d\n",ret);
   return 0;
}
```

GDB – Start Debug

- Start debug mode (`gdb gdbdemo`)
  - Note the executable (not .c files passed)
  - Note the last line before `(gdb)` prompt:
    - if –g option is not used while compiling, you will see "`(no debugging symbols found)`"

```
[ecegrid-thin4:~/ECE264] hegden$gdb gdbdemo
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/min/a/hegden/ECE264/gdbdemo...done.
(gdb)
```

GDB – Set breakpoints

- Set breakpoints (b)
  - At line 14
  - Beginning of foo

```c
1 #include<stdio.h>
2 int foo(int a, int b)
3 {
4      int x = a + 1;
5      int y = b + 2;
6      int sum = x + y;
7 
8      return x * y + sum;
9 }
10 
11 int main()
12 {
13     int ret = foo(10, 20);
14     printf("value returned from foo: %d\n",ret);
15     return 0;
16 }
```

```
(gdb) b gdbdemo.c:14
Breakpoint 1 at 0x400512: file gdbdemo.c, line 14.
(gdb) b foo
Breakpoint 2 at 0x4004ce: file gdbdemo.c, line 4.
(gdb) 
```

GDB – Manage breakpoints

- Display all breakpoints set (`info b`)

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400512 in main at gdbdemo.c:14
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Delete a breakpoint (d <breakpoint num>)

```
(gdb) d 1
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Disable a breakpoint (disable <breakpoint num>)

```
(gdb) disable 2
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep n   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Enable breakpoint (enable <breakpoint num>)

```
(gdb) enable 2
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
```

GDB – Start execution

- Start execution (r <command-line arguments>)
  - Execution stops at the first breakpoint encountered

```
(gdb) r
Starting program: /home/min/a/hegden/ECE264/gdbdemo

Breakpoint 3, main () at gdbdemo.c:13
13          int ret = foo(10, 20);
```

  - Continue execution (c)

```
(gdb) c
Continuing.

Program exited normally.
```

GDB – Step in

- Steps inside a function call (`s`)

```
Breakpoint 3, main () at gdbdemo.c:13
13          int ret = foo(10, 20);
(gdb) s
foo (a=10, b=20) at gdbdemo.c:4
4            int x = a + 1;
```

GDB – Step out

- Jump to return address (`finish`)

```
(gdb) finish
Run till exit from #0  foo (a=10, b=20) at gdbdemo.c:4
0x000000000040050f in main () at gdbdemo.c:13
13          int ret = foo(10, 20);
Value returned is $2 = 275
```

GDB – Printing

- Printing variable values (p <variable_name>)

```
Breakpoint 2, foo (a=10, b=20) at gdbdemo.c:4
4           int x = a + 1;
(gdb) n
5           int y = b + 2;
(gdb) p x
$3 = 11
```

- Printing addresses (p &<variable_name>)

```
(gdb) p &x
$5 = (int *) 0x7fffffffc4f4
```

GDB – Memory dump

- Printing memory content `(x/nfu <address>)`
  - n = repetition (number of bytes to display)
  - f = format ('x' – hexadecimal, 'd'-decimal, etc.)
  - u = unit ('b' – byte, 'h' – halfword/2 bytes, 'w' – word/4 bytes, 'g' – giga word/8 bytes)
  - `E.g. x/16xb 0x7fffffffc500` (display the values of 16 bytes stored from starting address 0x7..c500 and show them in hexa-decimal)

```
(gdb) x/16xb 0x7fffffffc500
0x7fffffffc500: 0x20    0xc5    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffc508: 0x0f    0x05    0x40    0x00    0x00    0x00    0x00    0x00
```

GDB – Printing addresses

- `Registers ($rsp, $rbp)`
  - Note that we use the ‘x’ command and not the ‘p’ command.

```
(gdb) x $rsp
0x7fffffffc500: 0x20
(gdb) x $rbp
0x7fffffffc500: 0x20
```

GDB – Altering memory content

- Set command (`set variable <name> = value`)

```
(gdb) n
6           int sum = x + y;
(gdb) p x
$7 = 11
(gdb) p y
$8 = 22
(gdb) set variable y = 0
(gdb) n
8           return x * y + sum;
(gdb) p sum
$9 = 11
```

- Set command (`set *(<type *>addr) = value`)

Buffer Overflow Attack

- Attacker gives input too big for a fixed-length buffer.
- When this has the effect of overwriting the return address, normal program execution is hijacked.
- When the return address is overwritten with starting address of a malicious code block (e.g. deleting all files), victim suffers.
- Example:
  - Ransomware WannaCry (2017/18) exploited buffer overflow vulnerability.

Buffer Overflow Attack

- To hijack the control:

1. First we need to identify the return address from a function.
2. Next, we need to identify the location (starting address) where the return address is stored.
3. Finally, we need to overwrite the contents at that location.
   a. Look for the location (address) where fixed-length buffers are stored.
   b. Compute offsets from that address that point to address identified in Step 2.
   c. Alter contents of the memory at those offsets.

Demo

Data Types

- What is a data type?
  - Way of indicating *what a variable is*.
  - Example:

```
int x;
```

1. *What is the set of values this variable can take on?*
2. *How much space does this variable take up?*
3. *How should operations on this variable be handled?*

```
int x;
```

1. *What is the set of values this variable can take on in C?*

   -2^{31} to $(2^{31} - 1)$

2. *How much space does this variable take up?*

   32 bits

3. *How should operations on this variable be handled?*

   integer division is different from floating point divisions

   ```
   3 / 2 = 1  //integer division
   ```

   ```
   3.0 / 2.0 = 1.5  //floating-point division
   ```

Data Types in C

- Basic
  - `int, char, float, double.`
- Modifiers
  - `short, long, signed, unsigned.`
- Compound types
  - pointers, structs, enums, arrays, etc.

Data Types in C – storage space

Data type	Number of bytes
`char`	1
`short int`	2
`int / long int`	4
`long long int`	8
`float`	4
`double`	8
`long double`	12

- Use sizeof() operator to check the size of a type
  - e.g. `sizeof(int)`

Data types - quirks

- if no type is given compiler automatically converts it to `int` data type.
  - `signed x;`
- `long` is the only modifier allowed with `double`
  - `long double y;`
- `signed` is the default modifier for `char` and `int`
- Can’t use any modifiers with `float`

Strings

- Array of `char`
- Terminated by the null character `‘\0’` as per convention
- Example:

```
char s[]=“ECE”;
```

s →	**‘E’**	**‘C’**	**‘E’**	**‘\0’**
Address	0x7fffc510	0x7fffc511	0x7fffc512	0x7fffc513
Value	69	67	69	0

Strings - Initializing

- char s1[3];
- s1[0]=‘H’; //ASCII 72
- s1[1]=‘i’; //ASCII 105
- s1[2]=‘\0’; //ASCII 0
- char s2[]=“Hi”;
- char s3[]={‘H’,’i’,’\0’};
- char* s4=“Hi”;
- char s5[]= {72, 105,0};
- char s6[]= {0x48, 0x69, 0}
- char s7[]=“\x48\x69”;

String Literals

- `String Literals`
- Example:
  - `printf(“Hello World\n”);`
  - `char *s =“Hi”;`
- On data segment (initialized)
  - Cannot modify them

is “Hi” a string literal here? `char s2[]=“Hi”`*;*

Exercise – Identifying memory segments (strings)

```
void oat(char pie)
{
   char ham;
   char bun[4];
   char* ice = “pop”;
   static char egg = 1;
   static char nut;
}

char jam = 2;
char tea;
```

parameter → Stack segment

Local variable → Stack segment

Statically allocated array / local variable → Stack segment

String literal → Data segment (read-only)

Address (still a local variable) → Stack segment

Static variable → Data segment (read-write)

Static variable → Data segment (uninitialized/ bss)

Global variable → Data segment (read-write)

Global variable → Data segment (uninitialized/bss)

String on stack and data segments

`char s[]="Aye";`	`char* s="Aye";`
can modify s: `s[0]='B';`	"Aye" is read-only. s[0]='B' is undefined behavior.
"Aye" is on stack segment	"Aye" is on data segment (initialized/read-only part)
Difficult to reassign s	Easy to reassign s: s = "Why"

- Print the length of a string using `strlen`

```
#include<string.h>
…
char s[]="Hello";
printf("%d\n",strlen(s));
```

- Use format specifier %s to print string values

```
printf("%s\n",s);
```

Arrays in C

Declaring arrays:

```
type <array_name>[<array_size>];
int num[5];
```

Initializing arrays:

```
int num[3]={2,6,4};
int num[]={2,6,4};//array_size is not required.
```

Accessing arrays:

num[0] accesses the first integer

num[1] accesses the second integer and so on..

Literals in C

- We saw string literals: “ECE”
- char literal: ‘E’, ‘C’
- int literal: 264
  - *is 018 an int literal?*
  - *What about 0xFee?*
- float literal: 3.142

Typedef

- Lets you give alternative names to C data types
- Example:

```
typedef unsigned char BYTE;
```

This gives the name BYTE to an `unsigned char` type. Now,

```
BYTE a;
BYTE b;
```

Are valid statements.

Typedef Syntax

```
typedef <existing_type> <new_type>;
```

- Resembles a declaration without initializer;

  E.g. `int x;`

- Mostly used with user-defined types

User-defined Types

- *Structures* in C are one way of defining your own type.
- Arrays are compound types but have the *same* type within.
  - E.g. A string is an array of `char`
  - `int arr[]={1,2,3};` `arr` is an array of integer types
- Structures let you compose types with *different* basic types within.

Structures

- Objectives:
  - How do we declare them?
  - How do we use them?
  - How do we initialize them?

Structures - Declaration

Type name

Declarations of fields

```c
struct Point{
float xCoordinate;
float yCoordinate;
};
```

- Variable definition:
  - `struct Point p1;`
  - ```c
    struct Point{
    float xCoordinate;
    float yCoordinate;
    }p1;
    ```

p1 is a variable (an object) of type `struct Point`
    ```

Structures - Definition

Declarations of fields

New Type name

```
typedef struct _Point{
    float xCoordinate;
    float yCoordinate;
}Point;
```

Canonical type name (long form)

- Variable definition:
  - `Point p1;`

Structures - Usage

- Structure fields are accessed using dot (.) operator
- Example:

```
Point p;
p.xCoordinate = 10.1;
p.yCoordinate = 22.8;
printf(“(x,y)=(%f,%f)\n”,p.xCoordinate,
p.yCoordinate);
```

Structures - Initialization

- Error to initialize fields in declaration;

```
typedef struct{
float xCoordinate = 10.1;
float yCoordinate = 22.8;
}Point;
```

Structures - Initialization

- `Point p1={10.1,22.8};`

- `Point p2={.x=10.1,.y=22.8};`

```
//Introduced in C99.
//Designated initializers
//Best-way
```

Structures - Exercise

```
typedef struct{
float x; //x Coordinate
float y; //y Coordinate
char name[4];
}Address;

int main()
{
Address p={.x=10.1, .y=20.2, .name=“WLPD”};
...
}
```

Is the structure object initialization okay?

Structures – Exercise 2

- *Can we have a structure as a field within a structure?*

```
typedef struct{
char street[128]; //street address
int zipCode;
}StreetNZip;

typedef struct{
float x; //x Coordinate
float y; //y Coordinate
char name[4];
StreetNZip detail;
}PointOnMap;
```

Homework1 Review

`alias`

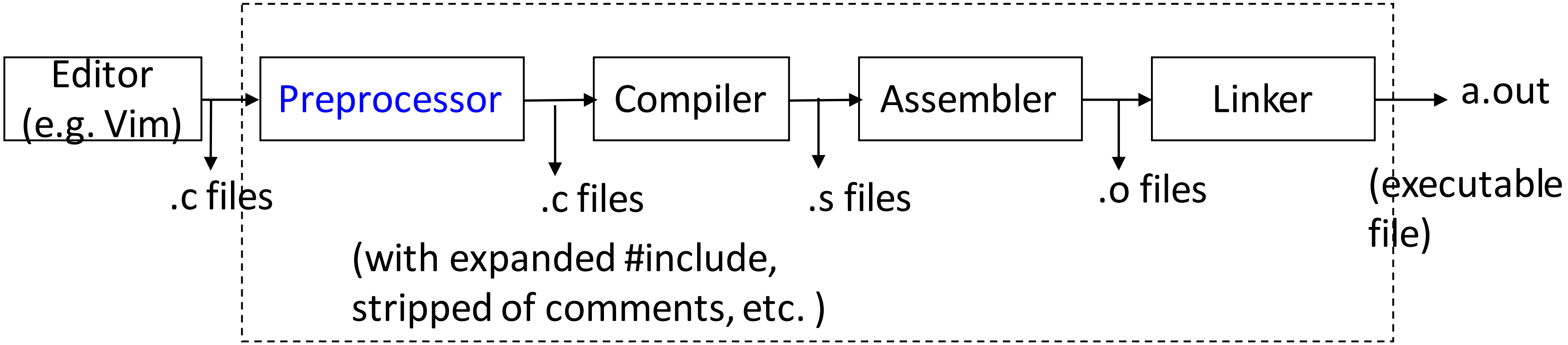
- `alias` for short forms of commands that are often used and are long to type
- Example
  - `alias gcc='gcc -std=c99 -g -Wall -Wvla –Werror --pedantic'`
- Not preserved across sessions
  - To preserve the command across terminal sessions, add the `alias` command to a file named `.bash_profile` located in the home directory (create the file if it does not exist).

Conditional Compilation

- Set of 6 preprocessor directives and an operator.
  - #if
  - #ifdef
  - #ifndef
  - #elif
  - #else
  - #endif
- Operator ‘defined’

Preprocessor - Detour

- Preprocessor – 1st step in transforming source code to an executable




```
gcc -std=c99 -g -Wvla -Werror -Wla --pedantic -Wall testgen.c
```

- Any line in source code that begins with '#' is a preprocessor directive.
  - #include, #pragma, #define, #ifdef

#if

```
#if <constant-expression>
printf("ECE264\n")
#endif
```

//The line containing printf is compiled only if `<constant-expression>` evaluates to a value > 0 while preprocessing

Example:

```
#define COMP 0
#if COMP
printf("ECE264\n")
#endif
```

No compiler error

```
#define COMP 2
#if COMP
printf("ECE264\n")
#endif
```

Compiler throws error about missing semicolon

#ifdef

```
#ifdef identifier
printf("ECE264\n")
#endif
```

//The line containing `printf` is compiled only if identifier is defined by the time the line with `#ifdef` is seen while preprocessing. Does not require a value to be set. Even if set, does not care about 0 or > 0.

Example:

```
#define COMP
#ifdef COMP
printf("ECE264\n")
#endif
```

```
#define COMP 0
#ifdef COMP
printf("ECE264\n")
#endif
```

```
#define COMP 2
#ifdef COMP
printf("ECE264\n")
#endif
```

All three snippets throw compiler error about missing semicolon

#else and #elif

```
1. #ifdef identifier1
2. printf(“summer\n”);
3. #elif identifier2
4. printf(“fall\n”);
5. #else
6. printf(“spring\n”);
7. #endif
```

//preprocessor checks if identifier1 is defined. if so, line 2 is compiled. If not, checks if identifier2 is defined. If identifier2 is defined, line 4 is compiled. Otherwise, line 6 is compiled.

`defined` operator

Example:

```
#if defined(COMP)
printf("spring\n");
#endif
```

//same as if #ifdef COMP

```
#if defined(COMP1) || defined(COMP2)
printf("spring\n");
#endif
```

//if either `COMP1` or `COMP2` is defined, the `printf` statement is compiled. As with `#ifdef`, `COMP1` or `COMP2` values are irrelevant.