SC3260 / SC5260

Crash course in C

Lecture by: Ana Gainaru

Slides of this lecture are adapted from Lewis Girod
CENS Systems Lab and Duke University C Crash Course
`https://people.duke.edu/~ccc14/sta-663/CrashCourseInC.html`

Why C/C++

Programming for High Performance

- Mainly because it produces code that runs nearly as fast as code written in assembly language
- There are many parallel programming languages build on top of C

VANDERBILT UNIVERSITY

Learning a Programming Language

"A little learning is a dangerous thing." (Alexander Pope)

This is a crash course

- The best way to learn is to write programs
- Take the examples from the previous lectures
- All you need is the gcc compiler (you can also use a virtual environment)
- Examples are provided on the course website

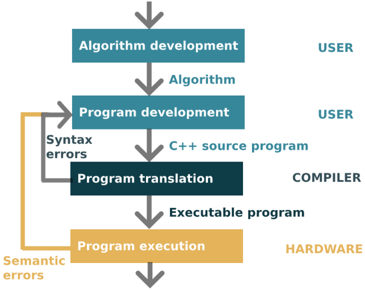
https://github.com/vanderbiltscl/SC3260_HPC

In the `C_code_examples` folder

VANDERBILT UNIVERSITY

Writing and Running Programs

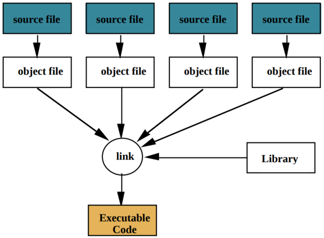
- Write text of program (source code) using an editor such as vim, eclipse, etc.
  - Save the file as a C program e.g. my_program.c
- Run the compiler to convert program from source to an executable **(List of instructions)**
  `gcc my_program.c -o my_program`
- Compiler gives errors and warnings; edit source file, fix it, and re-compile
- Run the executable
  `./my_program`



VANDERBILT UNIVERSITY

Writing and Running Programs

- **Editing**: Writing the source code by using some IDE or editor
- **Compiling** Translates source to object code for a specific platform
  - A compiler creates object code, which is an intermediary step between source code and final executable code
  - The compiler checks for syntax errors
    e.g., Missing punctuation
  - The compiler performs simple optimization on your code
    e.g., Eliminate a redundant statement
- **Linking** Links together all object modules to form an executable
  - If your program uses a library routine, like sqrt, the linker finds the object code corresponding to this routine and links it within the final executable
- **Loading Your Program** Loads your program into the computer memory
  - Performed automatically by the OS (except for embedded systems)



Hello World

```c
#include <stdio.h>

/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

- `#include` inserts another file. ".h" files are called "header" files. They contain stuff needed to interface to libraries and code in other ".c" files.
- The `main()` function is always where your program starts running.
- Blocks of code ("lexical scopes") are marked by `{ ... }`

VANDERBILT UNIVERSITY

Hello World

```
#include <stdio.h>

/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

- A Function is a series of instructions to run. You pass Arguments to a function and it returns a Value.
  - In our case the function returns an `int` (it can be `void` or a data type)
  - `printf()` is just another function, like `main()`.
    It's defined for you in a "library", a collection of functions you can call from your program (defined by `studio.h`).
  - The signature of the function needs to be declared

VANDERBILT UNIVERSITY

Comparison to Python

Compute Fibonacci numbers

The code will generate the x-th Fibonacci number. We don't need to store the entire sequence

fibonacci.py

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = a+b, a
    return a
```

main.py

```
import fibonacci
import sys

if __name__ == '__main__':
    n = int(sys.argv[1])
    print("%f" %(fibonacci.fib(n)))
```

fibonacci.c

```
double fib(int n)
{
    double a = 0, b = 1;
    for (int i=0; i<n; i++)
    {
        double tmp = b;
        b = a;
        a += tmp;
    }
    return a;
}
```

fibonacci.h

```
double fib(int n);
```

main.c

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

int main(int argc, char* argv[])
{
    int n = atoi(argv[1]);
    printf("%f\n", fib(n));
}
```

Comparison to Python

Greatest Common Denominator

Compute all numbers between 1 and n that are co-prime

gcd.py

```
def find_gcd(a, b):
  while True:
    if a > b:
      a = a - b
    elif a < b:
      b = b - a
    else:
      return a
```

gcd.c

```
int find_gcd(int a, int b)
{
  while (1)
  {
    if (a > b)
      a = a - b;
    else if ( a < b )
      b = b - a;
    else
      return a ;
  }
}
```

Comparison to Python

Greatest Common Denominator

Compute all numbers between 1 and n that are co-prime

gcd.py

```
def find_gcd(a, b):
  while True:
    if a > b:
      a = a - b
    elif a < b:
      b = b - a
    else:
      return a
```

python main.py 100
Execution time 0.016765356063842773 s to find 6007 coprime numbers
python main.py 1000
Execution time 4.019978046417236 s to find 607583 coprime numbers
python main.py 3000
Execution time 22.609691381454468 s to find 5470775 coprime numbers

gcd.c

```
int find_gcd(int a, int b)
{
  while (1)
  {
    if (a > b)
      a = a - b;
    else if ( a < b )
      b = b - a;
    else
      return a ;
  }
}
```

gcc -Wall main.c gcd.c -o gcd
./gcd 100
Execution time 0.000831 s to find 6007 coprime numbers
./gcd 1000
Execution time 0.111138 s to find 607583 coprime numbers
./gcd 10000
Execution time 16.458545 s to find 60786971 coprime numbers

VANDERBILT UNIVERSITY

Memory

- Imagine the memory like a big table of numbered slots where bytes can be stored.
  - A Type names a logical meaning to a span of memory. Some simple types are:

```
char          \\ a single character (1 byte)
char[10]      \\ an array of 10 characters
int           \\ signed 4 byte integer
double        \\ signed 8 byte integer
float         \\ 4 byte floating point
```

  - You can define your own type

```
#include <stdio.h>
struct point {
    double x;
    double y;
    double z;
};

typedef struct point point;

int main() {
    point p = {1, 2, 3};
    printf("%.2f,␣%.2f,␣%.2f", p.x, p.y, p.z);
};
```

Addr	Value
0	
1	
2	
3	
4	'H' (72)
5	'e' (101)
6	'l' (108)
7	'l' (108)
8	'o' (111)
9	'\n' (10)
10	'\0' (0)
11	
12	

VANDERBILT UNIVERSITY

Memory

- A **Variable** names a place in memory where you store a value of a certain **Type**
  - You first **Define** a variable by giving it a name and specifying the type, and optionally an initial value

```c
char x;
char y='e';

int z = 0x01020304;
```

Different types consume different amounts of memory. Most architectures store data on "word boundaries", or even multiples of the size of a primitive data type (int, char)

- Example: z will take 4 bytes in the memory

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	‘e’ (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

The compiler puts them somewhere in memory.

Lexical Scoping

Every Variable is defined within some scope.

A variable cannot be referenced by name from outside of that scope

Lexical scopes are defined with curly braces { }

- The scope of Function Arguments is the complete body of the function
- The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block
- The scope of Variables defined outside a function starts at the definition and ends at the end of the file (Global variables)

Is this different than python?

```c
void p(char x)
{
                /* p,x */
 char y;
                /* p,x,y */
 char z;
                /* p,x,y,z */
}
                /* p */
char z;
                /* p,z */

void q(char a)
{
 char b;
                /* p,z,q,a,b */
 while(1)
 {
    char c;
                /* p,z,q,a,b,c */
 }
 char d;
                /* p,z,q,a,b,d (not c) */
}

                /* p,z,q */
```

VANDERBILT UNIVERSITY

Boolean Expressions

- Python has built-in values (`True` and `False`) to evaluate boolean expressions
  - None, Zeros of any numeric type, Empty sequences and collections are all false

- **In C++, you can also use numeric values to indicate true or false.**
  - Anything that evaluates to 0 is considered false, while every other numeric value is true.

C++ Operator	Python Operator
`&&`	`and`
`||`	`or`
`!`	`not`
`&`	`&`
`|`	`|`

VANDERBILT UNIVERSITY

Expressions and Evaluation

Highest to lowest precedence (like in python)

- `()`
- `*, /, %`
- `+, -`

- The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit.

Note: Do not confuse & and &&
1 & 2 -> 0 whereas 1 && 2 -> <true>

VANDERBILT UNIVERSITY

Comparison and Mathematical Operators

Similar to every other programming language. A few differences:

Note the difference between ++x and x++

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

VANDERBILT UNIVERSITY

Comparison and Mathematical Operators

Similar to every other programming language. A few differences:

Note the difference between ++x and x++

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don't confuse = and ==

```
int x=5;
if (x==6) /* false */
{
 /* ... */
}
/* x is still 5 */
```

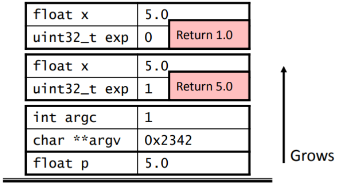
```
int x=5;
if (x=6) /* always true */
{
 /* x is now 6 */
}
/* ... */
```

The stack

Recall lexical scoping

- If a variable is valid "within the scope of a function", what happens when you call that function recursively?
- Is there more than one "exp"?

Yes. Each function call allocates a "stack frame" where Variables within that function's scope will reside.




```c
#include <stdio.h>
#include <inttypes.h>

float pow(float x, uint32_t exp)
{
  /* base case */
  if (exp == 0) {
    return 1.0;
  }

  /* "recursive" case */
  return x*pow(x, exp - 1);
}

int main(int argc, char **argv)
{
  float p;
  p = pow(5.0, 1);
  printf("p = %f\n", p);
  return 0;
}
```

VANDERBILT UNIVERSITY

Iterative vs recursive

```
float pow(float x, uint32_t exp)
{
 /* base case */
 if (exp == 0) {
   return 1.0;
 }

 /* recursive case */
 return x * pow(x, exp - 1);
}
```

Iterative using loops (while, for)

```
float pow(float x, uint exp)
{
 int i=0;
 float result = 1.0;
 while (i < exp) {
   result = result * x;
   i++;
 }
 return result;
}
```

Recursion eats stack space (in C). Each loop must allocate space for arguments and local variables, because each new call creates a new "scope"

VANDERBILT UNIVERSITY

Code syntax

Code syntax will not be covered (differences are small)

- Look at the example code. Any question about if/else/while/for/do?
- Homeworks will have starting code. Ask questions if the syntax is not clear !
- One difference example, the python ternary construct
  `expr = expr1 if condition else expr2`
  translates into C:

```
expr = condition ? expr1 : expr2
```

VANDERBILT UNIVERSITY

Code syntax

Code syntax will not be covered (differences are small)

- Look at the example code. Any question about if/else/while/for/do?
- Homeworks will have starting code. Ask questions if the syntax is not clear !
- One difference example, the python ternary construct
  `expr = expr1 if condition else expr2`
  translates into C:

```
expr = condition ? expr1 : expr2
```

Don't forget about the "whitespace issue"!

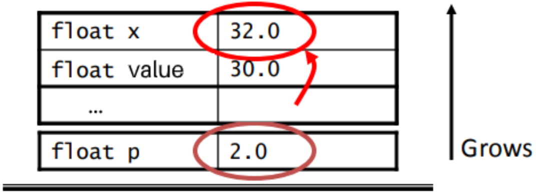
```
int i = 10;
while (i > 0)
    i--;
    i++;
```

VANDERBILT UNIVERSITY

Referencing Data from Other Scopes

Let's say we wanted to implement a function that modified its argument

```
void increment(float x, float value)
{
  x += value;
}

{
  float p = 2;
  increment(p, 30);
}
```



Referencing Data from Other Scopes

In Python there is the concept of mutable / unmutable objects

Examples

- **List** - a mutable type

```
def try_to_change_list_contents(the_list):
    the_list.append(4)
    print('changing␣to', the_list)

outer_list = [1, 2, 3]

print('before,␣outer_list␣=', outer_list)
try_to_change_list_contents(outer_list)
print('after,␣outer_list␣=', outer_list)
```

```
before, outer_list = [1, 2, 3]
got ['one', 'two', 'three']
changing to [1, 2, 3, 4]
after, outer_list = [1, 2, 3, 4]
```

- **String/Float/Int** - immutable types
  - Functions need to return the new value

VANDERBILT UNIVERSITY

Passing values by reference

- Every variable has a memory address associate with it

```
"address of variable" or reference operator: &
"value at address" or dereference operator: *
```

```c
void f(address_of_char p)
{
   value_at_address[p] = new_value;
}

char y = init_value;
f(address_of(y));

\\ y has the new value
```

```c
void f(char * p)
{
  *p = 'b';
}

char y = 'a';
f(&y);

\\ y is now 'b'
```

Addr	Value
0	
1	
2	
3	
4	'H' (72)
5	'e' (101)
6	'l' (108)
7	'l' (108)
8	'o' (111)
9	'\n' (10)
10	'\0' (0)
11	
12	

Pointers are used in C for many other purposes:

- Passing large objects without copying them
- Accessing dynamically allocated memory
- Referring to functions

VANDERBILT UNIVERSITY

Passing values by reference

Remember variables have alive only within a scope

```
char * get_pointer()
{
  char x=0;
  return &x;
}

{
  char * ptr = get_pointer();
  *ptr = 12; /* valid? */
  other_function();
}
```

VANDERBILT
UNIVERSITY

Passing values by reference

Remember variables have alive only within a scope

```
char * get_pointer()
{
  char x=0;
  return &x;
}

{
  char * ptr = get_pointer();
  *ptr = 12; /* valid? */
  other_function();
}
```

The pointer `ptr` will point to an area of the memory that may later get reused and rewritten.

Using invalid pointers will cause non-deterministic behavior,
and will often cause Linux to kill your process (SEGV or Segmentation Fault).

Arrays

```c
int dt[3] = {1, 2, 3};
char x[5] = "test";

/* accessing element 0 */
x[0] = 'T';

/* x[0] evaluates to the first element;
 * x evaluates to the address of the
 * first element, or &(x[0]) */

/* pointer arithmetic to get the third element */
char elt3 = *(x+3); /* x[3] == *(x+3) == 't' (NOT 's'!) */
```

Symbol	Addr	Value
char x [0]	100	‘t’
char x [1]	101	‘e’
char x [2]	102	‘s’
char x [3]	103	‘t’
char x [4]	104	‘\0’

Note: Arrays in C are 0-indexed

VANDERBILT UNIVERSITY

Dynamic Memory Allocation

All examples have allocated variables statically by defining them in our program. This allocates them in the stack.

```
int * alloc_ints(size_t requested_count)
{
  int * big_array;
  big_array = (int *)malloc(requested_count * sizeof(int));
  if (big_array == NULL)
  {
     printf("can't allocate %d ints\n", requested_count);
     return NULL;
  }

  /* now big_array[0] .. big_array[requested_count-1] are valid */
  return big_array;
}
```

It's OK to return this pointer. It will remain valid until it is freed with `free()`

The stack is automatically reclaimed. Dynamic allocations are not !
All objects must be tracked and freed when they are no longer needed.

Losing track of memory is called a "memory leak".

VANDERBILT UNIVERSITY

Dynamic Memory Allocation

A pointer is simply a name for an integer that represents an address
since it is an integer, it also has an address ...

```
// 2D arrays

// first allocate space for the pointers to all rows
int **arr = malloc(r * sizeof(int *));
// then allocate space for the number of columns in each row
for (int i=0; i<r; i++) {
    arr[i] = malloc(c * sizeof(int));
}

// fill array with integer values
for (int i = 0; i <  r; i++) {
    for (int j = 0; j < c; j++) {
         arr[i][j] =i*r+j;
    }
}

// every malloc should have a free to avoid memory leaks
for (int i=0; i<r; i++) {
    free(arr[i]);
}
free(arr);
```

```
// static allocation
int arr[r][c];
```

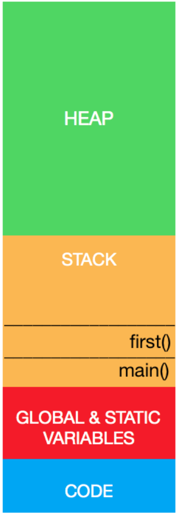
VANDERBILT UNIVERSITY

Example heap/stack

```c
int first(int a, int b){
  return a;
}

int main(){
        int a = first(5, 10);
        printf("%d\n", a);
}
```

- Operating system allocates a fixed memory for the stack
  - **Stack Overflow** if this is not enough
- Each function gets memory in the stack for the parameter, local variables, return values
  - This is called a **Stack Frame**
  - Size needs to be known at compile time
- The memory allocation/deallocation is handled by the OS

Let's look at an example



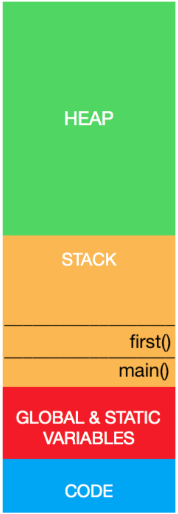
Example heap/stack

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

int maine(){
   int a;

   int *p;
   p = (int *) malloc (10 * sizeof(int));
   *p = 10;
   free(p);

   int *addr_a = &a;
   *addr_a = 6;
}
```

- The heap is also called **Dynamic memory** and it's handled by the user
  - Memory leak when dynamic memory is not deallocated

Is this code correct?

```c
   p = (int *) malloc (sizeof(int));
   *p = 10;

   p = (int *) malloc (sizeof(int));
   *p = 20;
   free(p);
```



VANDERBILT UNIVERSITY

Let's look at some examples

- Basic examples

  https://github.com/vanderbiltscl/SC3260_HPC

  In the `C_code_examples` folder

- Buggy example

  In the `C_code_examples/buggy` folder

- Building our own
  - Depending on time, we will look at matrix tiling and k-step Fibonacci

VANDERBILT UNIVERSITY

Notes of linking

Including `<stdlib.h>` or `<stdio.h>` in a C program doesn't need linking when compiling but `<math.h>` needs `-lm` with `gcc`

- The functions in `stdlib.h` and `stdio.h` have implementations in `libc.so` (or `libc.a`for static linking), which is linked into your executable by default (as if `-lc` were specified). GCC can be instructed to avoid this automatic link with the `-nostdlib` or `-nodefaultlibs` options.
- The math functions in `math.h` have implementations in `libm.so` (or `libm.a` for static linking), and `libm` is not linked in by default.

VANDERBILT UNIVERSITY

Debugging a C program

When a C program crashes

- Return a **segmentation fault** message
- There is no indication on where in the C program the error occurred

- Compile your C program with the debug flag -g

```
gcc -g  prog-file1.c
gcc -g  prog-file2.c
...
gcc -o yourProgName prog-file1.o prog-file2.o
```

- Use dbg to debug the error

```
gdb yourProgName

//At the gdb prompt, type the run command:
(gdb) run
```

Debugging a C program

```
anagainaru@VUSE-5W6CBH2:~/work/lectures/2020_hpc/code$ gdb a.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb) run
Starting program: /home/anagainaru/work/lectures/2020_hpc/code/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005c0 in main (argc=1, argv=0x7fffffffde48) at sigfault.c:8
8              a[i] = 1234;
(gdb) where
#0  0x00000000004005c0 in main (argc=1, argv=0x7fffffffde48) at sigfault.c:8
(gdb) list
3       int main(int argc, char *argv[])
4       {
5              int a[10];
6              int i = -872625577;
7
8              a[i] = 1234;
9              printf("i = %d\n", i);
10             printf("a[i] = %d\n", a[i]);
11      }
(gdb) p i
$1 = -872625577
(gdb) p a[i]
Cannot access memory at address 0x7fff2ff3068c
(gdb) quit
```

VANDERBILT UNIVERSITY

Debugging a C program

```
anagainaru@VUSE-5W6CBH2:~/work/lectures/2020_hpc/code$ gdb a.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb) run
Starting program: /home/anagainaru/work/lectures/2020_hpc/code/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005c0 in main (argc=1, argv=0x7fffffffde48) at sigfault.c:8
8              a[i] = 1234;
(gdb) where
#0  0x00000000004005c0 in main (argc=1, argv=0x7fffffffde48) at sigfault.c:8
(gdb) list
3       int main(int argc, char *argv[])
4       {
5              int a[10];
6              int i = -872625577;
7
8              a[i] = 1234;
9              printf("i = %d\n", i);
10             printf("a[i] = %d\n", a[i]);
11      }
(gdb) p i
$1 = -872625577
(gdb) p a[i]
Cannot access memory at address 0x7fff2ff3068c
(gdb) quit
```

- You can create your own break points
- **Does not always give enough information**
- When in doubt, use `printf()`

Further reading

"A little learning is a dangerous thing." (Alexander Pope)

- These slides should cover everything you need to know to understand the examples in this class
- **Does not** cover everything there is to know about C
- Run all the examples we give you on ACCRE

Books to learn C++

1. **A Tour of C++** or **The C++ Programming Language** by Bjarne Stroustrup
2. **C++ in a nutshell** by Ray Lischner
   pdf available here: https://github.com/vpreethamkashyap/Library

VANDERBILT UNIVERSITY