SCALABLE COMPUTING LABORATORY

VANDERBILT UNIVERSITY

SC3260 / SC5260

Introduction to C

Lecture by: Ana Gainaru

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

Many slides of this lecture are adapted from Lewis Girod, CENS Systems Lab http://lecs.cs.ucla.edu/ girod/talks/c-tutorial.ppt and Clark Barrett

Learning a Programming Language

This is a crash course

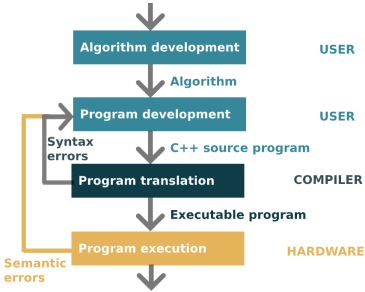
- The best way to learn is to write programs
- Take the examples from the previous lectures
- All you need is the gcc compiler (you can also use a virtual environment)
- Examples are provided on the course website

https://github.com/vanderbiltscl/SC3260_HPC

In the `C_code_examples` folder

Writing and Running Programs

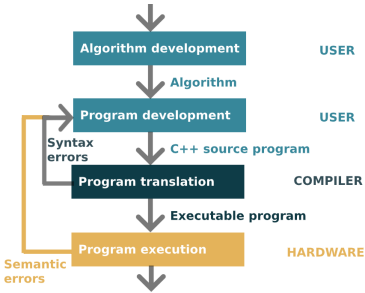
- Write text of program (source code) using an editor such as vim, eclipse, etc.
  - Save the file as a C program e.g. my_program.c
- Run the compiler to convert program from source to an executable **(List of instructions)**
  `gcc -Wall my_program.c -o my_program`
- Compiler gives errors and warnings;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



VANDERBILT UNIVERSITY

Writing and Running Programs

- Hardware independent
- Programs portable to most computers
- Case-sensitive
- Four stages
  - **Editing**: Writing the source code by using some IDE or editor
  - **Preprocessing or libraries**: Already available routines
  - **Compiling**: translates or converts source to object code for a specific platform (source code -> object)
  - **Linking**: resolves external references and produces the executable module



VANDERBILT UNIVERSITY

Compiling

Compilation occurs in two steps: Preprocessing and Compiling

In **Preprocessing**, the source code is expanded into a larger form that is simpler for the compiler to understand.

- Any line that starts with # is a line that is interpreted by the Preprocessor.
- Include files are pasted in (`#include`)
- Macros are "expanded" (`#define`)
- Comments are stripped out (/* */ , //)
- Continued lines are joined (\ )

During **Compilation** the resulting text is converted into binary code the CPU can run directly

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 `{ ... }`

Hello World

```c
#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)
  - C`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

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 slot)
char[10]      \\ an array of 10 characters
int           \\ signed 4 byte integer
float         \\ 4 byte floating point
int64_t       \\ signed 8 byte integer
```

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	

Memory

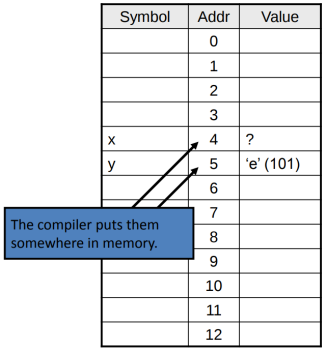
- A **Variable** names a place in memory where you store a value of a certain **Type**
  - You first **Define** a variable by giving it a name and specifying the type, and optionally an initial value

```
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



VANDERBILT UNIVERSITY

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)**

```
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

Expressions and Evaluation

Expressions combine Values using Operators, according to precedence

```
1 + 2 * 2       -->     1 + 4   -->     5
(1 + 2) * 2     -->     3 * 2   -->     6
```

- Comparison operators are used to compare values.
  - In C, 0 means "false", and any other value means "true".

```
int x=4;
(x < 5) -->     (4 < 5) -->     <true>
(x < 4) -->     (4 < 4) -->      0

((x < 5) || (x < 4))    -->     (<true> || (x < 4))     -->     <true>
```

Note that `x < 4` will not be evaluated

Highest to lowest precedence

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

- 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>

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 */
```

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 */
```

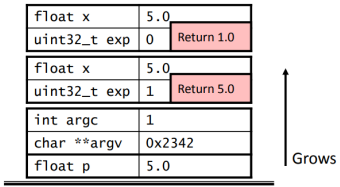
```
int x=5;
if (x=6) /* always true */
{
 /* x is now 6 */
}
/* ... */
```

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"