# CAAM 420/520: Programming in C

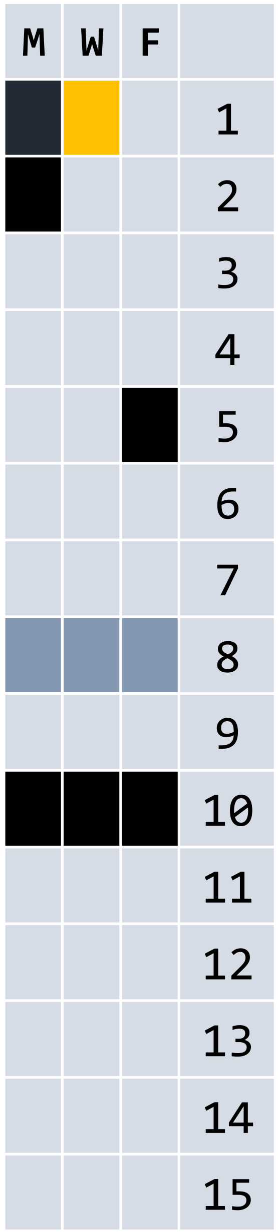
Section: Pre-Parallelism

Date: 1/11/2023

~~M: Course and Syllabus Overview~~

**W: Programming in C/C++**

F: Compiling and Running C/C++ Code



# Updates:

- Office Hours:
  - Wednesdays 2-3pm
  - ~~Fridays 2-3pm~~
  - Fridays 11am-12pm
- Applications are open for ATPESC:
  - Argonne (NL) Training Program for Extreme Scale Computing
  - Deadline: March 1
  - https://extremecomputingtraining.anl.gov/?ct=t(EVT-ATPESC2023_01032023)

# In these slides:

- Basic Syntax
  - Statements, control flow, and code blocks
- Logic and Operators
- Variables and Arrays
- Pointers
- Functions

# Syntax: Statements must end with a ';'

**C/C++**

```
x = 4  // Compiler will complain
x = 4; // Compiler happy
x = 4; y = 5; // Don't do; see Code Standards
```

**Code Standards:** one statement per line

**MatLab**

```
x = 4  % Will print
x = 4; % Won't print
x = 4; y = 5;
```

**Julia**

```
x = 4  # Prints if last
x = 4; # Won't print
x = 4; y = 5;
```

**Python**

```
x = 4  # Does not care
x = 4; # Still ok
x = 4; y = 5
```

# Syntax: Single and multi-line comments

**C/C++**

```
// Single line comment
/* Multiple line comment part 1
   part 2 ..................................... */
func(x, weirdVar /* because… */);
```

Multiline comments can also be embedded in code

**MatLab**

```
% Single line only :(
```

**Julia**

```
# Single line
#= No one knows this
exists? =#
```

**Python**

```
# Single line only :(
```

# Syntax: Parentheses around conditionals

Conditionals are the statements for control flow

**C/C++**

```
if( x > 4 ) {
     // code
}
```

**MatLab**

```
if x > 4
  % code
end
```

**Julia**

```
if x > 4
  # code
end
```

**Python**

```
if x > 4
  # code
# Python use indent
```

# Syntax: Braces around code blocks

True for if’s/else if’s/else, loops (while, for, do-while)

**C/C++**

```
if( x > 4 ) {
      // code
 }
if( x > 10 )
      x = 5;
```

- **Exception:** one-liners
- **Code Standards:** open brace on the same line as the conditional

**MatLab (cannot do)**

```
if x > 4
  % code
end
```

**Julia (cannot do)**

```
if x > 4
  # code
end
```

**Python (doesn’t matter)**

```
if x > 4
  # code
# Python use indent
```

# Syntax: `if, else-if, else` Statements

**C/C++**

```
if( condition1 ) {
   action1();
} else if( condition2 ) {
   action2();
} else {
   default_action();
}
```

**Code Standards:**
else-if/else’s on the same line as the previous block’s ‘}’

# Logic: Comparison Operators

- **Equality:** x == y **Common error:** x = y (assignment)!
- **Not equal:** x != y
- **Less than:** x < y
- **Less than or equal:** x <= y
- **Greater than:** x > y
- **Greater than or equal:** x >= y

**Note:** Comparisons with floating point numbers will be enforced <u>exactly</u> (to be addressed later)

# Logic: Predicate Logic Operators

- **Negation/Not:** !
- **And:** && (2 ampersands)
- **Or:** || (2 pipe symbols)
- **Bitwise and:** & (1 ampersand)
  - Also used for reference variables and addressing
- **Bitwise or:** | (1 pipe)
- **Bitwise xor (exclusive or):** ^
  - There is no exponentiation operator

**Code Standards:** Do not use bitwise operators without permission

# Predicate Logic

**de Morgan’s Law:**

```
!(A && B) = !A || !B
!(A || B) = !A && !B
```

i.e., the negation of an “and” is an “or” and vice versa

**Strive to make your conditionals as simple to understand as possible**

- Positive statements (is) are often (but not always!) easier to understand than negative statements (is not)
- Try to simplify compound statements if possible

# Predicate Logic

**de Morgan's Law:** also true for several statements

$$!(A_1 \ \&\& \ A_2 \ \&\& \ \ldots \ \&\& \ A_n) = \ !A_1 || !A_2 \ || \ \ldots \ || !A_n$$

$$!(A_1 \ || A_2 \ || \ \ldots \ || \ A_n) = \ !A_1 \ \&\& \ !A_2 \ \&\& \ \ldots \ \&\& \ !A_n$$

Special Note:

- **Nesting if-statements acts like an "and"**
- However, the nesting can change the branching (i.e. what outcomes are possible)

# Predicate Logic: Nested if’s

These statements are equivalent

```
if( x < 5 ) {
    if( y < 4 ) {
        // Do something
    }
}
```

```
if( x < 5 && y < 4 ) {
    // Do something
}
```

# Predicate Logic: Nested if’s

**These statements are NOT equivalent**

```
if( x < 5 ) {
   if( y < 4 ) {
      // Do something
   } else {
      // Do other thing
   }
}
```

**Here, in the “else” you know that x < 5 and y >= 4**

```
if( x < 5 && y < 4 ) {
   // Do something
} else {
   // Do other thing
}
```

**Here, you do not know which condition(s) failed in the “else”**

# Predicate Logic: Nested if’s

**These statements are equivalent again**

```
if( x < 5 ) {
    if( y < 4 ) {
        // Do something
    } else {
        // Do other thing
    }
}
```

```
if( x < 5 && y < 4 ) {
    // Do something
} else if( x < 5 ) {
    // Do other thing
}
```

**Try to keep your statements as simple as possible!**

# Short Circuiting: And’s

**“And”s need <u>all</u> arguments to be true**

- If the first argument is <u>false</u>, why check the others?
- **Short circuit:** in a string of “and”s, once one condition evaluates to false, the rest are not checked

**Example:** `while( i < n && x[i] < target )`

- If `length(x) = n`, then `x[i]` could crash if `i >= n`
- **Short circuiting can protect fallible statements**

# Short Circuiting: Or’s

**“Or”s need just one argument to be true**

- If the first argument is true, why check the others?
- **Short circuit:** in a string of “or”s, once one condition evaluates to true, the rest are not checked

**Example: `if( error(x) < tol || max(x) > ub )`**

- Some conditions may be expensive to evaluate
- **Short circuiting can save time (and maybe memory)**

# Assignment Operators

- **Increment/decrement by one:**
  - **Prefix:** `++i; --i;` (change, then evaluate)
  - **Postfix:** `i++; i--;` (evaluate, then change) <- **preferred**
- **Increment:** `x += 4;`
- **Decrement:** `x -= 4;`
- **Multiplication:** `x *= 4;`
- **Division:** `x /= 4;`

**Can also do bitwise operators:** `|=, &=, ^=`

# The Modulus Operator, %

- **The modulus operator returns the remainder of the argument vs the base:**

`5 % 2 = 1,     8 % 3 = 2,     15 % 4 = 3`

- It is very helpful when dealing with multi-dim arrays…
- **Note:** `abs(n % m)` will always be in the range [0, m-1]
- You can use negative integers too

# Variables: Declaring vs defining

C/C++

```
int x;          // Declares the variable x
x = 4;          // Initializes/defines x
double y = 4; // Declares and initializes y
```

MatLab

```
x = 4;   % Integer
y = 4.0; % Floating pt
```

Julia

```
x = 4   # Integer
y = 4.0 # Floating pt
```

Python

```
x = 4   # Integer
y = 4.0 # Floating pt
```

# Primitive Variable Types

- **Integer Types:**
  - **int:** Integer variables; used to represent whole numbers
  - **char:** Characters; used to represent text. A char variable holds 1 character (i.e. one letter, digit, or symbol). [0,255]
  - **bool:** Boolean variables; can take the value 0 (false) or 1 (true)
  - **Pointers:** Pointers are more or less just hexadecimal (base 16) integers
- **Floating Point Types: float** and **double**
  - doubles have twice (i.e. double) the memory of floats
- **Modifiers:** do not work on all types
  - **short** (16bit; int), **long** (32bit; int, double), **long long** (64bit; int)
  - **unsigned:** (int, char); doubles the variable's range of values
  - **signed:** (char); for using chars are numbers only

# Expectations

**You are expected to know:**

- **Floating point arithmetic is not exact**
- **Integer division truncates, it does not round**
- If you are not familiar with these things, please do some research on them

# Variables: Structures

**C/C++**

```
struct TypeName {
    int x;
    double y;
}; // Note the weird semicolon

TypeName var; // Declares an instance
```

**Note:** There are many different syntaxes for declaring structs. **You should not need to declare any types for your homeworks;** any special types will be given already. If you do declare your own type, use the above syntax.

# Variable Size in Memory

**The amount of memory each variable has can vary with the system and compiler**

**Always the same:**

- **1 bit = a 0 or a 1;** the fundamental unit of memory
- **1 byte = 8 bits**

**The system decides the size of a "word" of memory:**

- **16bit (really old):** 2 bytes per word = 16 bits
- **32bit (older but still around):** 4 bytes per word = 32 bits
- **64bit:** 8 bytes per word = 64 bits

# Variable Size in Memory

**All you really need to know about variable sizes:**

char <= short int <= int <= long int <= long long int

float < double <= long double

**Typically:**

- **char:** 8bit
- **int:** 32bit
- **long long int:** 64bit
- **float:** 32bit (7-8 digits of precision)
- **double:** 64bit (15-16 digits of precision)

**DO NOT ASSUME memory size, use sizeof()**

# Reading and Printing Variables

## We will often use printf and (maybe) scanf

- printf and scanf use what are called format specifiers
  - Format specifiers allow you to write to (scanf) and print (printf) variables
- Common format specifiers are:
  - %d (or i) – Signed integers (please use d)
  - %f, %lf – Floats (%f) and doubles (%lf)
  - %c – characters
  - %s – strings (C style = array of chars)
  - Uncommon: %p – pointer, %u – uint, %o – octal uint, %x – hexadecimal uint
- printf:
  - %.2f -> prints 2 digits after the decimal point, can put any (integer) number
  - %e – Prints in scientific notation; can also do %.3e like with %f
- scanf: variables need a ‘&’ in front, pointers do not
- Can do more things; good reference: cplusplus.com, search printf

# Variables: Declaring Arrays

C/C++

```
int x[5]; // Declares an array of 5 ints
          // with unknown values

// Declares and initializes the array
int y[] = {1, 2, 3, 4, 5};
// Zero-indexing!!! y[0] = 1, y[4] = 5!
```

**What if we don't know the size we need yet?**

=> Dynamic memory allocation and <u>pointers</u>

# Variables: Dynamic Mem. Allocation

C++

```
int* x = NULL;
int* array = NULL;

// Allocates a single int and an array of ints
x = new int;
array = new int[<number of array elements>];

// Must free memory!
delete x;
delete[] array;
```

This is the C++ way of allocating memory

# Variables: Dynamic Mem. Allocation

C

```c
int* x = NULL;
// Allocates an array dynamically
x = (int*) malloc(sizeof(int)*<num elem>);

// Must free memory!
free(x);
```

This is the C way of allocating memory

# Different Base Systems

## A quantity in any representation is the same quantity

| | 10 | 2 | 3 | 4 | 16 |
|---|---|---|---|---|---|
| -NOTHING- | 0 | 0 | 0 | 0 | 0 |
| O | 1 | ($2^0$) 1 | 1 | 1 | 1 |
| OO | 2 | ($2^1$) 10 | 2 | 2 | 2 |
| OOO | 3 | 11 | ($3^1$) 10 | 3 | 3 |
| OOOO | 4 | ($2^2$) 100 | 11 | ($4^1$) 10 | 4 |
| O OOOO | 5 | 101 | 12 | 11 | 5 |
| OOOO OOOO | 8 | ($2^3$) 1000 | 22 | (2x4) 20 | 8 |
| OO OOOO OOOO | 10 | 1010 | 101 | 22 | A |
| | 11 | 1011 | 102 | 23 | B |
| | 16 | ($2^4$) 10000 | 121 | ($4^2$) 100 | ($16^1$) 10 |

# Pointers: What are they?

**Pointers are (more or less) hexadecimal (base 16) integers**

- **Base 16?**
  - Aka hex/hexadecimal.
  - Decimal is base 10 (what we are familiar with)
  - Binary is base 2
  - Octal is base 8
  - **Trivia:** vigesimal (base 20) was used by the indigenous people of Mesoamerica

# Why Hex?

**Hexadecimal numbers can encode more information in fewer digits than smaller bases**

| Decimal (10) | Binary (2) | Hexadecimal (16) |
|---|---|---|
| 1 | 0000 0001 | 1 |
| 2 | 0000 0010 | 2 |
| 3 | 0000 0011 | 3 |
| 8 | 0000 1000 | 8 |
| 15 | 0000 1111 | F |
| 16 | 0001 0000 | 10 |
| 255 | 1111 1111 | FF |

# Memory: Bits

**00001010100101010001010001010010…**

# Memory: ~~Bits~~ -> Bytes

**00001010 10010101 00010100 01010010…**

**1 byte = 8 bits (8 binary digits)**

# Memory: ~~Bits~~ -> Nibbles

1
0x00000001

32
0x00000020

**0000 1010 1001 0101 0001 0100 0101 0010…**

15
0x0000000f

# Memory: Back to Bits

**Computers have a lot of memory, and the memory is "aligned" with power of 2**

**00001010100101010001010001010010…**

**01001010010100010100000010101000…**

**01010101001010101010000001001000…**

**00010100101111001000101001000001…**

# Why Represent Pointers in Hex?

- **They can encode more info in fewer digits**
- Modern computers have gigabytes ($10^9$ bytes) of RAM
- Bases that are powers of 2 can interface with binary more easily
- Memory is aligned according to powers of 2

# …Then why do pointers need the var type?

- **Why isn't there a type "pointer"? (vs "int* ptr;")**
- If we want to access the variable at the address given, the computer needs to know how much to grab
- Arrays: how big a step should the computer take?

| Bytes -> | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| int (32 bits) | | | | | | | | | | | | |
| double (64 bits) | | | | | | | | | | | | |
| Struct (int + double) | | | | | | | | | | | | |
| int[3] | | | | | | | | | | | | |

# Functions

## C/C++

```
<return type> name(<args>) {
    return …;
}

double add(double x, double y) {
    return x + y;
}
```

## MatLab

```
function […] = name(<args>)
end # return statement not needed!
f = @(x,y) x + y;
```

## Julia

```
function name(<args>)
    return …
end
f(x,y) = x + y;
```

## Python

```
def name(<args>)
    return …
```

# Functions: Void returns

C/C++

```
// Do not need a return statement
void func() { // No return or args
    printf(“Hello\n”);
}

// Can use one to short circuit
void func() {
    printf(“Hello\n”);
    return;
    printf(“This won’t print :(\n”);
}
```

# Functions: Pass by Value

C/C++

```
// Will not change x in main
void func(int x /*<-this is a new var declaration*/){
   x = 4;
}

int main() {
    int x = 5;
    func(x);
    // x will still be 5
    return 0;
}
```

# Functions: Pass by Reference

C/C++

```
// This WILL change x in main
void func(int &x /*<-this uses the existing x*/){
   x = 4;
}

int main() {
    int x = 5;
    func(x);
    // x will now be 4
    return 0;
}
```

# Functions: Pass by Pointer

C/C++

```
// This WILL change x in main
void func(int* x /*<-this declares a pointer…*/){
    *x = 4; // Use * to dereference a pointer
    // i.e., access the contents at the address
}

int main() {
    int x = 5;
    func(&x); // You have to pass x’s address
    // x will now be 4
    return 0;
}
```

# Functions: Passing Arrays

C/C++

```
// Arrays are actually pointers
void func(int A[] /*<-still declares a pointer…*/){
    A[2] = 4; // Use like normal
    A[3] = 8;
}

int main() {
    int A[] = {1, 2, 3, 4};
    func(A);
    // A will now be {1, 2, 4, 8}
    return 0;
}
```

# Functions: Passing Arrays

C/C++

```
// Arrays are actually pointers
void func(int* A){
    A[2] = 4; // Use like an array
    A[3] = 8;
}

int main() {
    int A[] = {1, 2, 3, 4};
    func(A); // No '&' needed! A is already a ptr
    // A will now be {1, 2, 4, 8}
    return 0;
}
```

# Pointers and Structs

C/C++

```
struct Point {
    double x;
    double y;
};
// To access struct fields from pointers, use "->" not "."
int main() {
     Point p;
     Point* p_ptr = new Point;

     p_ptr->x = 4;
     p.x = 4;
     … // clean up and return
}
```