CIS 330
C++ and Unix

Lecture 2

Intro to C (I/O and Operators)

Logistics

COVID containment plans for classes

https://provost.uoregon.edu/covid-containment-plan-classes

Prevention: To prevent or reduce the spread of COVID-19 in classrooms and on campus, all students and employees must:

- Comply with vaccination policy
    a. University of Oregon students and employees are **required** to be **fully vaccinated** against COVID-19 prior to the start of the fall academic term.
- Wear face coverings in all indoor spaces on UO campus
- Complete weekly testing if not fully vaccinated or exempted
- Wash hands frequently and practice social distancing when possible
- Complete daily self-checks
- Say home/do not come to campus if feeling symptomatic
- Complete the UO COVID-19 case and contact reporting form if you test positive or have been in close contact with a confirmed or presumptive case.

Logistics

Containment: If a student in class tests positive for COVID-19, everyone should :

- Expect and follow guidance in classroom notification
- Answer the call if contact by the Corona Corps (541-356-2292)
- Isolate if you test positive or are symptomatic
- Quarantine if you are a close contact
- Test weekly if you are unvaccinated or partially vaccinated
- Stay home if symptomatic and complete the Complete the UO COVID-19 case and contact reporting form

Logistics

Good Classroom Citizenship

- Wear your *mask* and make sure it fits you well
- *Stay home* if you're sick
- *Get to know your neighbors* in class, and let them know if you test positive
- *Get tested* regularly
- Watch for *signs and symptoms* with the daily symptom self-check
- *Wash your hands* frequently or use hand sanitizer
- Complete the UO COVID-19 *case and contact reporting form* if you test positive or are a close contact of someone who tests positive.

Logistics

Support: The following resources are available to students.

- University Health Services or call (541) 346-2770
- University Counseling Center or call (541) 346-3277 or (541) 346-3227 (after hrs.)
- MAP Covid-19 Testing
- Corona Corps or call (541) 346-2292
- Academic Advising or call (541) 346-3211
- Dean of Students or call (541)-346-3216

Logistics

Other things:

- Be courteous to others
- Eating and drinking is not allowed in class
- Any issues (e.g., non-compliance to masks) may result in a cancelled class (with an optional make-up class)

Course Website & Info

- jeewhanchoi.com/uocis330w22/
- Description
  - Practical software design and programming activities in a **C/C++** and **Unix** environment, with emphasis on the details of C/C++ and good programming style/practices.
- Prerequisite
  - CIS 314 (Computer Organization)

Instructor & TA

- Instructor
  - Jeewhan Choi (Jee)
- TA
  - Aktilek Zhumadil
- Office Hours
  - Instructor
    - Tuesday, Thursday 10:00 – 11:00 ***and by appointment***
  - TA
    - Tuesday, Thursday 11:00 – 1:00

Textbook

- Physical textbook is not required for this class
- See the class website for online text and resources

Grading

Criteria	Percentage
Homework	40%
Lab	10%
Quiz	10%
Midterm	20%
Final	20%

Grading

Score	Letter Grade
97 - 100	A+
93 - 96	A
90 - 92	A-
87 - 89	B+
83 - 86	B
80 - 82	B-
77 - 79	C+
73 - 76	C
70 - 72	C-
67 - 69	D+
63 - 66	D:
60 - 62	D-
< 60	F

Scores will NOT be rounded up. For example, 96.9 is a A.

Homework

- Assigned every Wednesday
  - Due the following Wednesday, 11:59 PM PST (usually)
- Lab attendance is required - grade will be based on lab submission & attendance
  - One lowest lab grade will be dropped
- Quiz will be given randomly on the materials covered in the previous class
  - Solutions will be discussed during class
- Submission of any homework/lab/quiz will **NOT** be allowed after it has been discussed in class

Grading

- All homework will be graded on **functionality** and **aesthetics**
  - Proper use of **comments**, white space, **indentation**, intuitive variable names, etc.
- **Code that does not compile** will be given 0
  - Must compile with "-std=c11" for C code, and "-std=c++14" for C++ code
  - Must compile and run on ix-dev (with the software available for everyone on the system)
- **Late homework will not be graded**, except
  - prior arrangement have been made *at least* 24 hours prior to the due date, or
  - **documented** emergencies
- Use version control
  - e.g., one single large commit may be subject to point deduction
- Develop code in Unix/Linux environment
  - e.g., any sign of Visual Basic or non-Unix/Linux environment may be subject to point deduction

Exams

Midterm

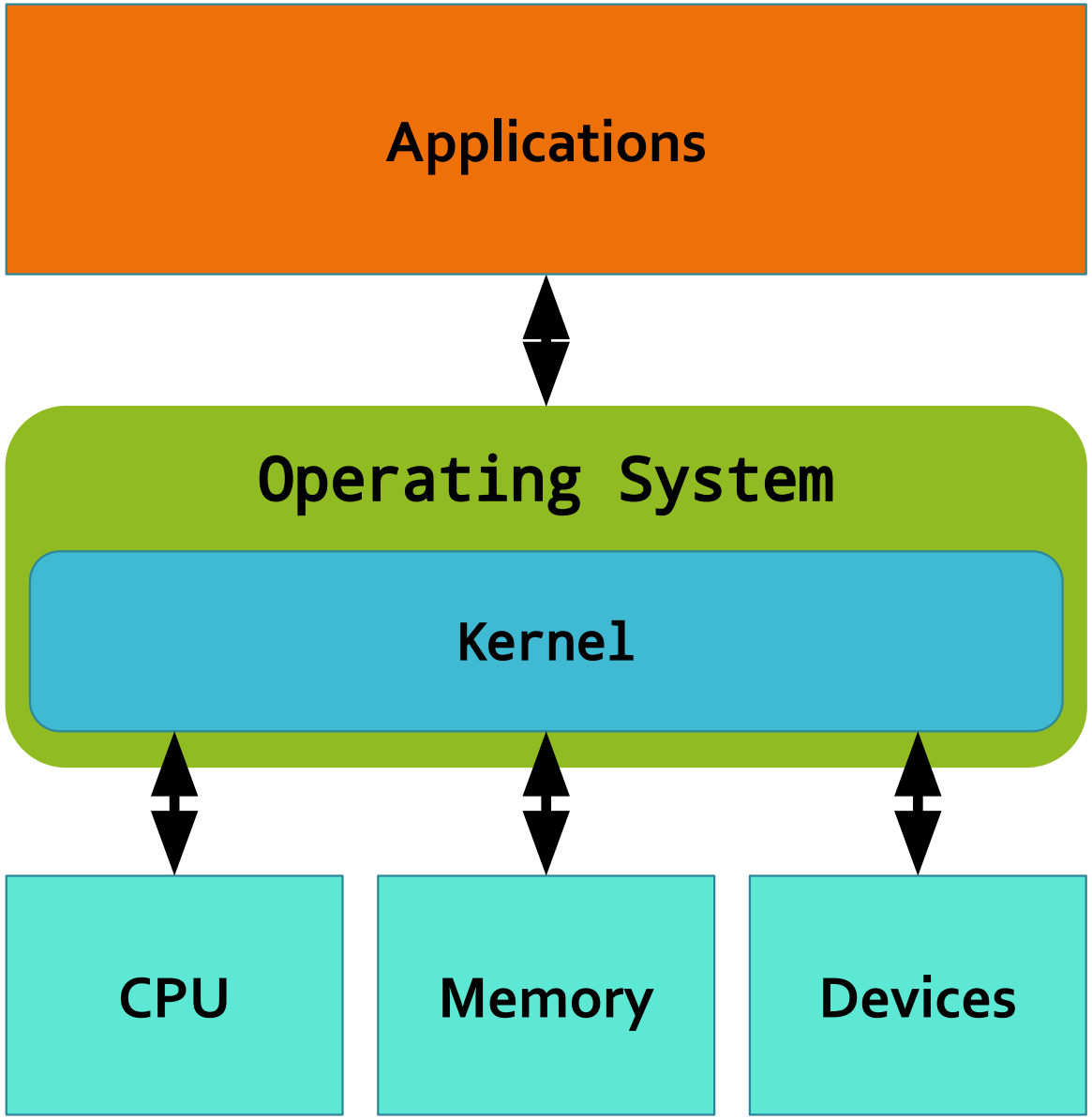
- Tuesday, Feb. 1st (week 5)

Final

- Monday Mar. 14 8:00 - 10:00 (2 hours)

Questions?

Last lecture - the Unix and Linux OS



Version Control Systems (VCS)

- A method for tracking changes to files
- A way to work collaboratively
- A way to maintain a centralized or distributed shared copy of projects

Why is it useful?

Individuals

- Backups
- Incremental versions (can **revert**)
- Tagging
- Branching
- Complete change **history**

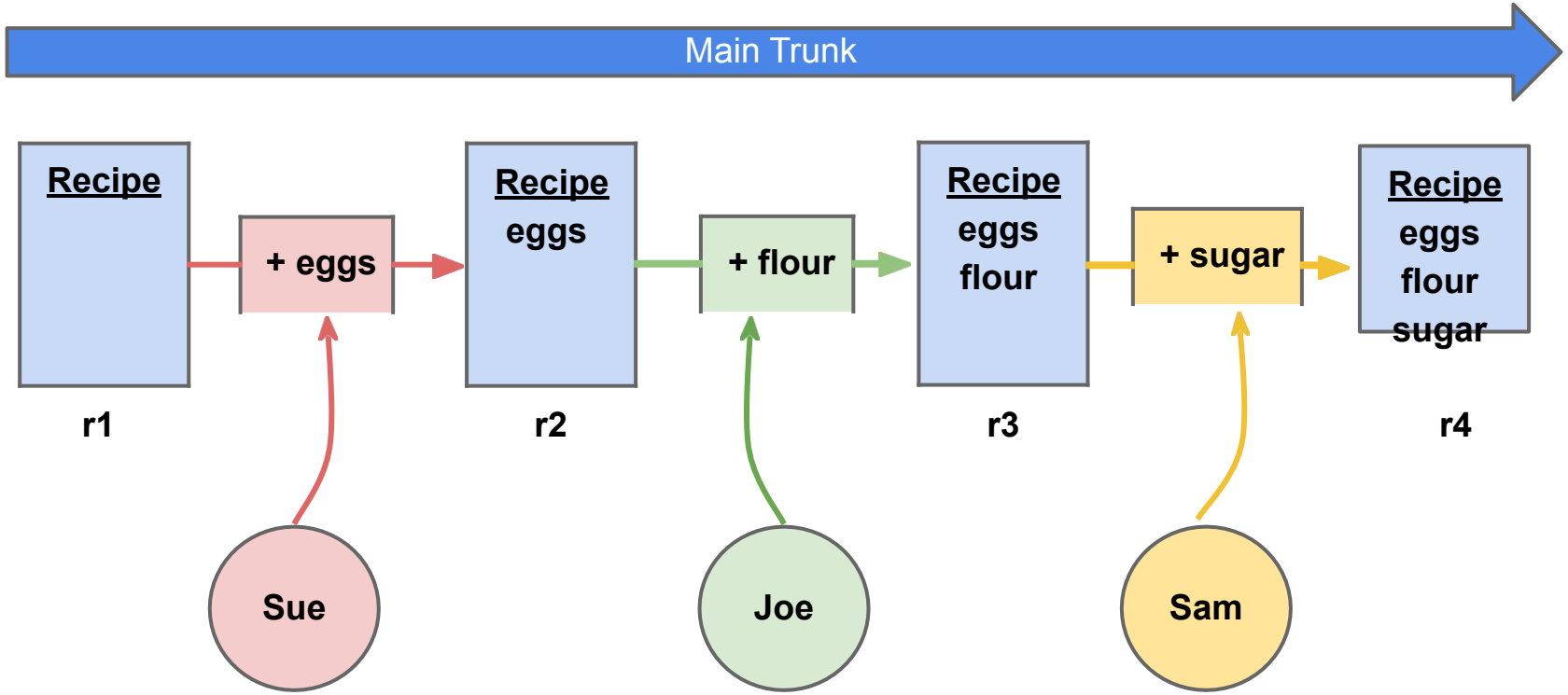
Groups

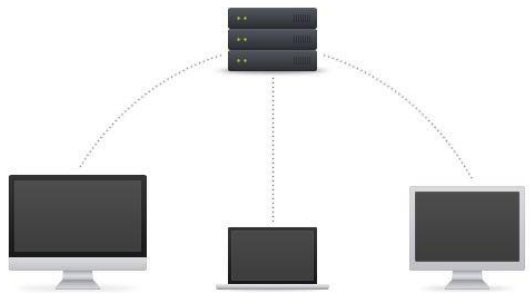
- Same as individuals
- Allow multiple developers to work on the same project
- Merge changes and handle conflicts
- Accurately assign blame

Types

- **Centralized**
- **Distributed**

Centralized

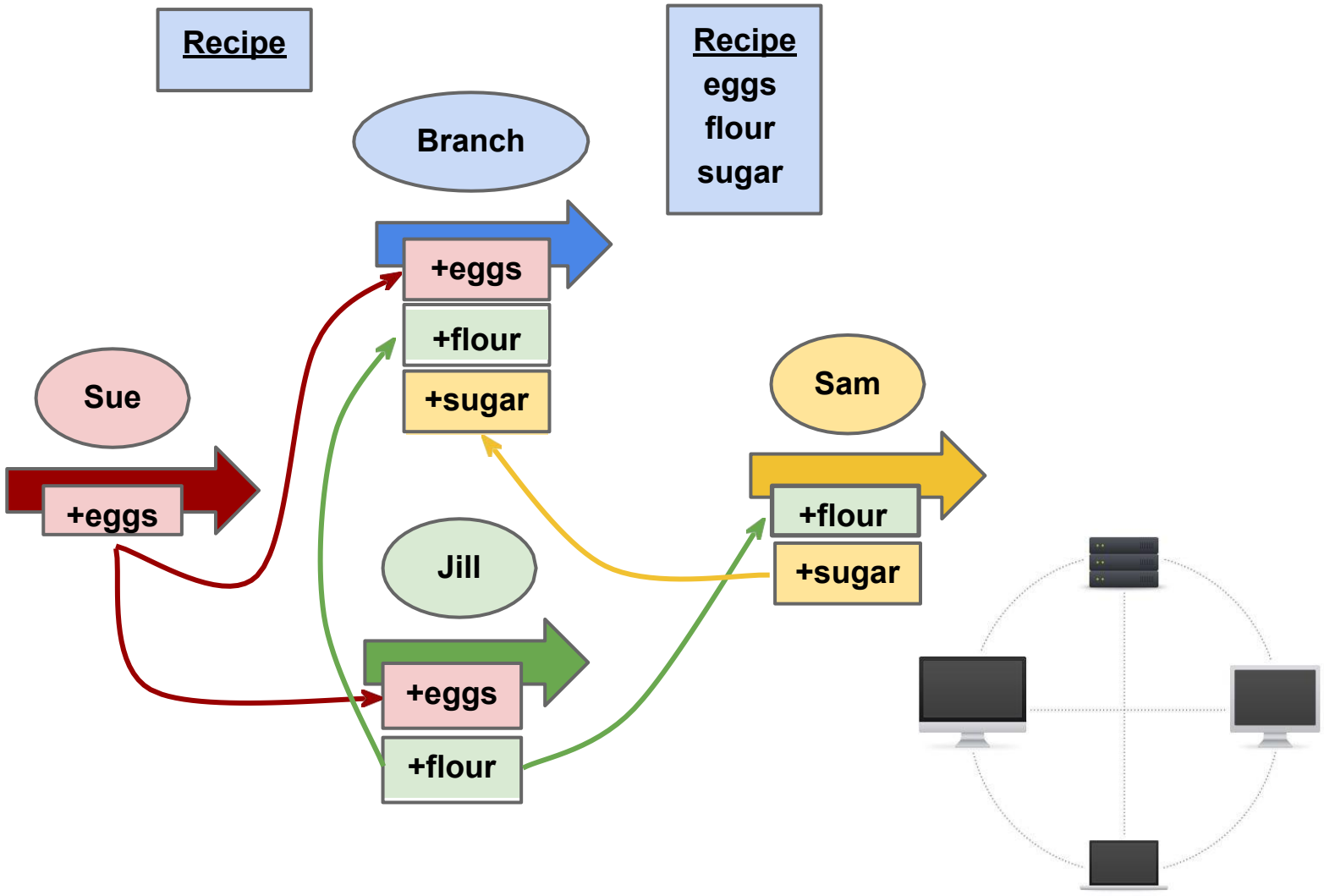




Examples of Centralized VCS

- **Concurrent Version System (CVS)**
- **Apache Subversion (SVN)**

Distributed



Examples of Distributed VCS (DVCS)

- **Git (not Jit)**
- **Mercurial**
- **Bazaar**

Centralized vs. Distributed

Centralized

- Does not requires extra space (especially if you have big binaries)
- Easier to keep track of changes

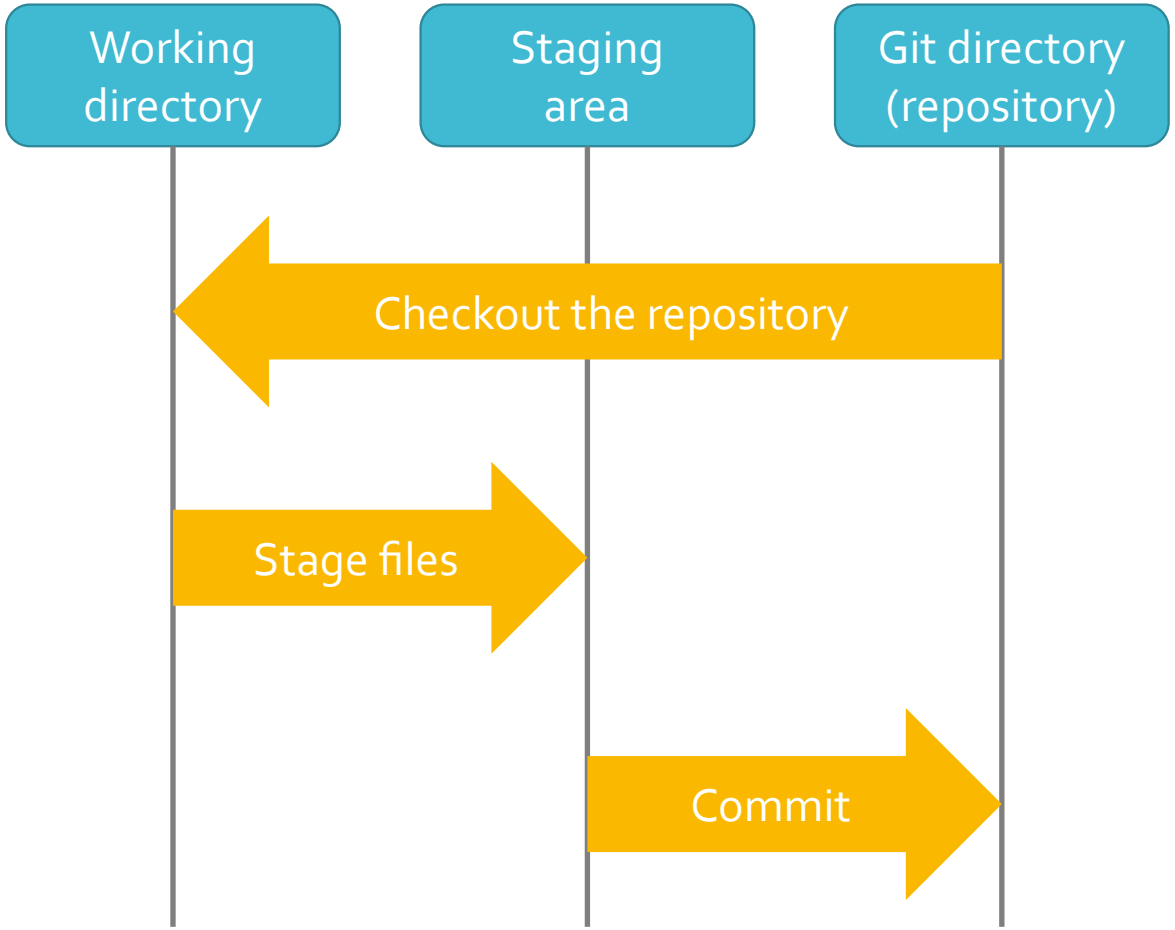
Distributed

- Faster to commit changes (less communication to the central repository)
- No single point of failure (each copy is “complete”)
- Available offline

Reading

- Git Book Chapters 1-3

Git workflow



Example

```
> git init
Initialized empty Git repository in
/gpfs/projects/hpctensor/jeec/class/2019/S19/CIS330/.git/
> git add lecture02.c
> git commit -m "first lecture code"
> git log
commit dfaa4e86109a427e32783b6ed72a4793b91f3d65
Author: Jee Choi <jeec@talapas-ln1.cm.cluster>
Date:   Fri Mar 15 18:34:16 2019 -0700

    first lecture code
```

Do not add binaries and object files to your repo - just the code and other files necessary to compiling your code

Do not add swap files (e.g., when you are using vim, it creates .swp files)

Always make sure to read all messages from git

- If you have a merge conflict, fix it and make sure the code compiles before you submit your assignment

Always keep committing your code at regular intervals and push it to Bitbucket

- If something goes wrong with your repo, you can always clone it and start from the last "save point."

Live demonstration of using Git (if time allows)

Questions?

The C Language

- Functional program – a program consists of a set of functions
- Each function consists of a set of statements
  - Each statement ends with a semicolon
  - Comments can be added to describe your statements, functions, etc.
    - `// This is a comment` (comments a line)
    - `/* This is also a comment */` (can span multiple lines)
- **Please comment your code!**
  - Believe me, you won't remember what you wrote after *X* month

Simple C Program

```c
#include <stdio.h>   /* Needed for printf and getchar */

int main()
{
    char userInput;

    printf("Are you having a nice day? (y/n)? ");

    userInput = getchar();

    if (userInput == 'y') {
        printf("That's wonderful, so am I!\n");
    } else {
        printf("That sucks, just try again tomorrow\n");
    }

    return 0;
}
```

Compiling and Running Your Code

- gcc is a compiler that converts your **source code** to **machine code** so that the processor can execute it (e.g., GNU compiler collection, or **gcc**)
- It generates an **executable** that you can run (typically **a.out** in Linux)

> gcc –std=c11 main.c

> ./a.out

Data Types

- Integers – **int**
- Character – **char**
- Floating point numbers – **float, double**
  - float -> single precision
  - double -> double precision
  - Since you can't have infinite precision (you would need infinite amount of memory), you use different number of bits to represent decimal numbers.
  - Consist of significand (e.g., 6.667) and exponent (e.g., -11) to represent the final value (e.g., 0.00000000006667).
- You can also "extend" the range of values
  - long int
  - long double
- Data can be "stored" in variables
  - Variable names are made up of letters, numbers, and _
  - However, it cannot begin with a digit

Special characters

- '\n' newline
- '\t' tab
- '\\' backslash
- And more…

Standard input & output (I/O)

- Three pre-defined I/O "streams":
  - stdin (standard input – i.e., keyboard)
  - stdout (standard output – i.e., screen)
  - stderr (standard error – i.e., screen)
  - There are others…
- printf
  - Prints a string to the standard out
- fprintf
  - You can specify which stream to write to
  - `fprintf(stdout, "That's wonderful, so am I!\n");`
- getchar
  - Get a **char** from the standard input
- scanf
  - Get an input from the standard input
  - `scanf("%c", &userInput);`
  - There is also **fscanf**

Standard input & output (I/O)

- How do you print the content of a variable?
- `fprintf(stdout, "5 + 5 is %d\n", some_number);`
- %d indicates that the variable you want to print here is an integer
- %f -> floating point number
- %c -> character
- %s -> string (more on this later)
- Works similarly for input

File I/O

```
FILE* fp = fopen("example.txt", "r");
char line[100];
while(fgets(line, 100, fp) != NULL) {
    fprintf(stdout, "%s", line);
}
fclose(fp);
```

Arithmetic

- Addition
  - + (binary)
  - ++ (unary)
  - **e.g., i = k++ -> i = k; k = k + 1;**
  - **e.g., i = ++k -> k = k + 1; i = k;**
- Subtraction
  - - (binary)
  - -- (unary)
- Multiplication
  - * (binary)
- Division
  - / (binary)
- Modulus (remainder)
  - % (binary)
  - **e.g., i = 10 % 3 -> i = 1**
- There are also ternary operators (i.e., operations with three variables)

Logical Operators

- && – Logical AND
- || – Logical OR
- ! – Logical NOT

Bit-wise Operators

- & - AND
  - e.g., 0101 & 1110 -> 0100
- | - OR
- ^ - XOR
- ~ - 1's complement
  - Flip each bit
- << - Shift left
- >> - Shift right
  - Do not use shift operators on negative or very large numbers (unpredictable behavior)
  - Equivalent to multiply/divide by 2
  - Faster than division/multiplication

Relational Operators

- Equality
  - ==
  - If (a == b) { /* Do something if they are equal */ }
- Inequality
  - !=
  - If (a != b) { /* Do something if they are NOT equal */ }
- Greater/Less than
  - >, >=, <, <=

Assignment

- a = b /* A gets the value of b */
- Combinations
  - +=, -=, *=, /=, &=, >>=, <<=, &=, ^=, !=
  - a += b -> a = a+ b

Control Flow

- if
- switch
- Repetition
  - while
  - do
  - for
- Branching
  - break
  - continue
  - goto

if conditional

```
1.  int i = 1;
2.  int j = 2;
3.
4.  if(i == j) {
5.      printf("Hello\n");
6.  } else {
7.      printf("Good bye\n");
8.  }
```

if conditional

```
1.  int i = 1;
2.  int j = 2;
3.
4.  if(i == j) {
5.      printf("Hello\n");
6.  } else if(i <= j) {
7.      printf(”Hola\n");
8.  }  else {
9.      printf("Good bye\n");
10. }
```

if conditional (ternary)

1. a ? b : c -> if a is true then the expression evaluates to b, otherwise to c
2. (i == j) **?** printf(“A\n”) **:** printf(“B\n”);

switch

```
1.  int input_int;
2.  fscanf(stdin, "%d", &input_int);
3.  switch (input_int + 1) {
4.      case 1:
5.          printf("You entered '0'\n");
6.          break;
7.      case 2:
8.          printf("You entered '1'\n");
9.          break;
10.     default:
11.         printf("What did you enter?\n");
12. }
```

while

```
1.  int i = 0;
2.  while(i < 10) {
3.    fprintf(stdout, "i is currently %d\n",i);
4.    i++;
5.  }
```

do-while

```
1.  int i = 0;
2.  do {
3.    fprintf(stdout, "i is currently %d\n",i);
4.    i++;
5.  } while(i < 10);
```

for

```
1.  int i = 100;
2.  for(i = 0; i < 10; i++) {
3.    fprintf(stdout, "i is currently %d\n",i);
4.  }
```

Macros

- #define
- #define PI 3.14 /* Define a constant value */
- #define calcCircleArea(r) (3.14 * (r) * (r)) /* Calculate area */
- Macros "calls" are replaced in the source code before compilation (by the C preprocessor)
- Similarly for #include and header files
  - May include other #define and function declarations

Functions

- Return a value (int/char/etc. function)
- Not return a value (void function)

int/char/float function

```
1.  int add_two_numbers(int a, int b)
2.  {
3.      return a + b;
4.  }
```

void function

```
1.  void add_two_numbers(int a, int b)
2.  {
3.      fprintf(stdout, "%d + %d = %d\n", a, b, (a + b));
4.  }
```

Main function parameters

```
1.  #include <stdio.h>
2.  #include <stdlib.h>

3.  int main(int argc, char **argv)
4.  {
5.      int cnt = 0;
6.      printf("argc == %d\n", argc);
7.      for(int i = 0; i < argc; i++) {
8.          printf("%s\n", argv[i]);
9.      }
10.     return 0;
11. }
```

```
> ./a.out this is good
```

Main function parameters

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

int main(int argc, char **argv)
{
    int cnt = 0;
    printf("argc == %d\n", argc);
    for(int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

```
> ./a.out this is good
argc == 4
./a.out
this
is
good
```

typedef

1. `typedef float ftype;`

- Why?
  - Keeps the code "independent" of data type
    - You can switch from using float to double by changing just the typedef
    - Or you can find & replace float with double using vim
  - Keep data type name easier to handle
    - unsigned long long int -> ullint

Compound data structures

```
1.  struct human_struct {
2.      char name[200];
3.      int age;
4.  };
5.  struct human_struct human_a;
```

- Data type with multiple entries – sometimes it make sense to associate multiple data types to a particular type of entity

Accessing individual components of a struct

```
1. strcpy(human_a.name, "The Doctor");
2. human_a.age = 1000;
```

typedef with struct

```
1.  typedef struct human_struct_2 {
2.      char* name;
3.      int age;
4.  } human_t;
5.  human_t human_b;
```

- or, if you don't need the struct to have a name

```
typedef struct {
    char species[200];
    char name[200];
    int age;
} being_t;
```

Copying Struct

- Makes a "shallow" copy – individual components are copied one at a time
- What happens if you have pointers?

Pointer to struct

```
1.  typedef struct {
2.      char species[200];
3.      char name[200];
4.      int age;
5.  } being_t;
6.  being_t* real_human_a;
7.  real_human_a = malloc(sizeof(being_t));
8.  strcpy(real_human_a->species, "Human");
9.  strcpy(real_human_a->species, "Rory Williams");
10. real_human_a->age = 23;
```

- Use “->” to access the struct components

Enum

```
1.  enum Color1 {
2.      red,
3.      green,
4.      blue
5.  };
6.  enum Color1 c1 = red;
```

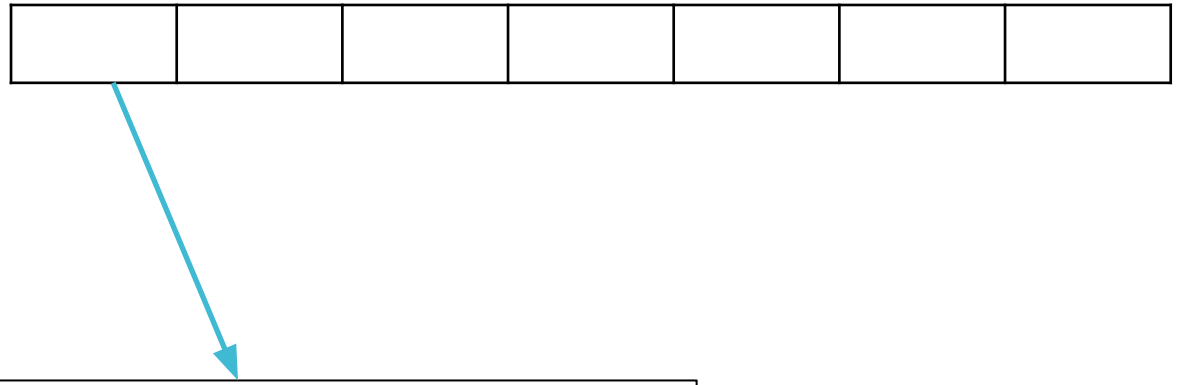
- Enumerated data type – each variable can store of the three options specified above

Union

```
1.  typedef union number {
2.      int i;
3.      float f;
4.      char *str;
5.  } number_t;
```

- Data type that allows a variable to take different types of values
- The members all reside in the same memory location – **not** stored redundantly

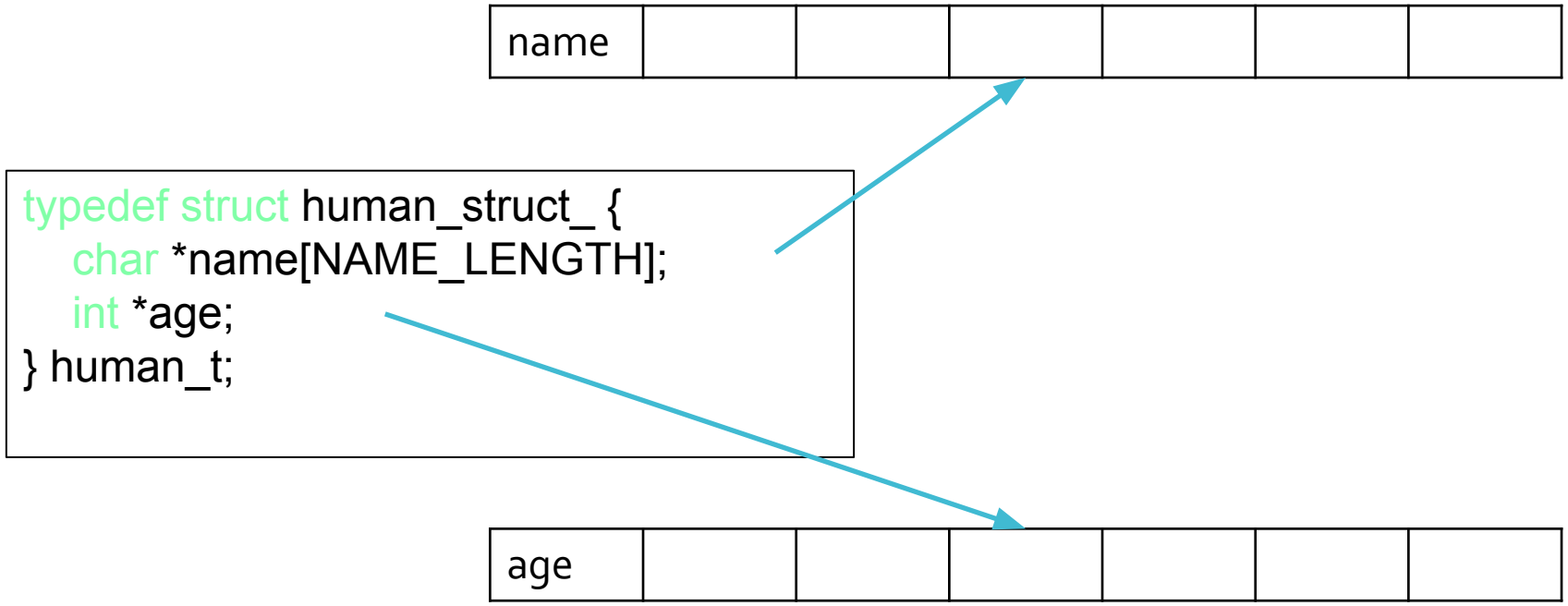
Array of Structure (AoS)



```
typedef struct human_struct_ {
    char name[NAME_LENGTH];
    int age;
} human_t;
```

- Array of structures – each element in the array is a structure
- Data for each structure is stored consecutively

Structure of Arrays (SoA)



- There is a single structure, where each element in the structure is an array
- Each element is store consecutively – good when you want to access a group of elements (e.g., searching for a name)

Code used in class

- Will be uploaded to bitbucket (after cleaning it up a bit)