CS343

Day 1

- For assignments, copy *uIO.cc*

> Why is concurrency important?
>
> - for job interviews
> - processor speed has slowed even though Moore's Law still holds true (number of transistors is still doubling every 18 months), so we need to stamp many cores into a single computer and run programs concurrently and in parallel
>
> Why C++?
>
> - C++ is a dominant programming language
> - large programmer and code base
> - efficient memory management while still maintaining high level features like OOP, exception handling, etc.
>
> Concurrency in C++
>
> - First C++ didn't have concurrency
> - It gained concurrency features over time
> - There is still no de facto approach dominating concurrency in c++
> - Therefore we us uC++ which is better than C++ in terms of having a standard concurrency support
> - Concurrency can't be safely added to any language with library code after the fact!
>
> High Level Concurrency
>
> - C++ supports some concurrency models better, which applies to all languages
> - C++ has OOP so we will use an object oriented concurrency model
>
> uC++
>
> - Integrates advanced control flow into C++
>   - Class-based/lightweight, communication with routine call, statically typed
> - Supports multiple processors

Advanced Control Flow

Milti-exit loop checks to exit at each iteration

```
for (;;) {
  if (...) break;
}
```

Eliminates priming with *while*:

```
cin >> d;
while (! cin.fail()) {
  ...
  cin >> d;
}
```

versus

```
for (;;) {
  cin >> d;
  if (cin.fail()) break;
}
```

Preferences

- We care more about readability, avoid complex structure
- An exit if clause should *never* have an else clause
- Do minimal nesting
- Avoid *flagitis*!
- Avoid using *goto*'s manually!
- Sometime's flag variables are necessary but at most 1 or 2 are ever needed, and that is rare
- Avoid duplication

Labelled break restrictions

- Don't use it to go backward in the program (don't create loops with it), only go forward
- Don't use it to go into a control structure, only use it to exit or stay in the same structure

Dynamic Memory Allocation

- You can allocate memory for variables on stack or heap
- Don't put everything on the heap, use the stack when you can!

Good

```
{
  cin >> size;
  int arr[size];
  // use arr
}
```

Bad

```
{
  cin >> size;
  int * arr = new int[size];
  // use arr
  delete [] arr; // why "[]"?
}
```

When do we use the heap?

- When a variable's storage must outlive the block in which it is allocated

  ```
  Type * rtn() {
    Type * tp = new Type;
    // ...
    return tp;
  }
  ```

- When the amount of data read is unknown

  ```
  vector<int> inputl
  int temp;
  for (;;) {
    cin >> temp;
    if (cin.fail()) break;
    input.push_back(temp); // implicit dynamic allocation
  }
  ```

  - Does `emplace_back` help? No, all it does is remove the copy call.
  - As programmers, we must know when we use dynamically sized data structures as that data is stored on the heap
- When an array of objects must be initialized via the object's constructor

  ```
  Object * objs[size];
  for (...) {
    objs[...] = new Obj(...); // each element has different value
  }
  ```

- When large local variables are allocated on a small stack

```
int arr[1000000]; // overflow
// vs
int * arr = new int[1000000];
```

Dynamic Multi-Level Exit

- *Modularization*: contiguous code that can be refactored into a helper
- Sometimes with labelled exits, we can't modularize our code because the label or exit may move into a different subroutine
-