Penn Engineering
University of Pennsylvania

CIS 3990 Recitation 06

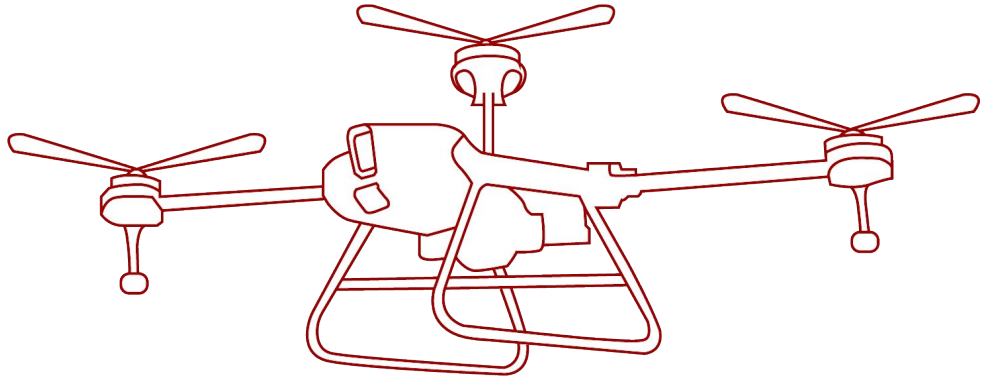
Midterm review

2025-10-16

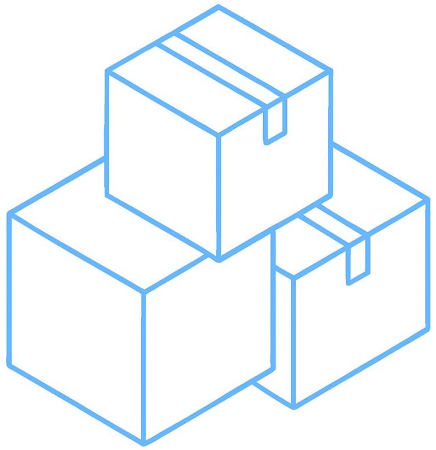
Agenda

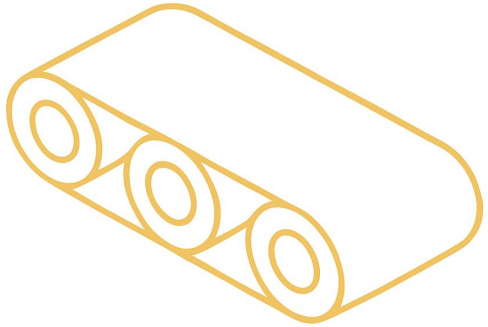
01	Course logistics
02	Midterm logistics
03	Content review & practice
04	Open Q&A

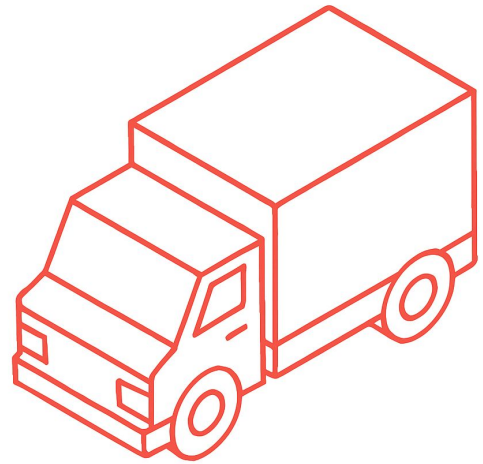
Penn Engineering
University of Pennsylvania



Logistics







Logistics

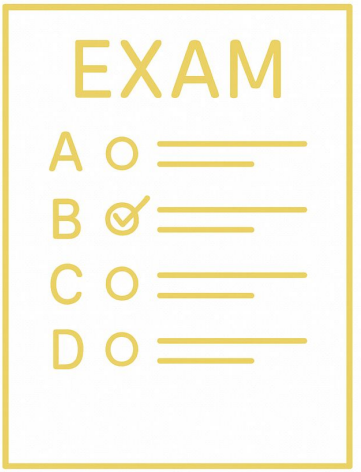
- HW06 out - due 11:59pm on **Fri, Oct 17** - via Gradescope
- Re-opens for the last check-in are processed, let me know if forgot anything/ selected the wrong date/assignment
- Remember HW06 is graded for style!
- Check-in due Mon, Oct 20 is posted

Logistics

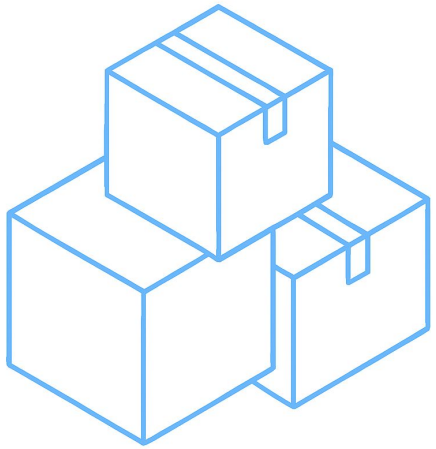
- Every past hw graded as of last night, including resubmissions

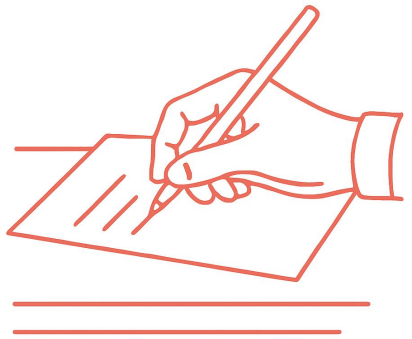
HW05: git [REPO CREATION]	0.0	OCT 1, 2025 2:00 PM	OCT 7, 2025 11:59 PM Late Due Date: DEC 8, 2025 11:59 PM	15	100%
HW05: git	100.0	OCT 1, 2025 2:00 PM	OCT 7, 2025 11:59 PM	12	100%
HW04: File Readers	250.0	SEP 23, 2025 11:59 PM	SEP 30, 2025 11:59 PM	10	100%
HW03: NFA	300.0	SEP 17, 2025 5:00 PM	SEP 25, 2025 11:59 PM	13	100%
HW02: Finite State Machine	200.0	SEP 10, 2025 2:00 PM	SEP 16, 2025 11:59 PM	15	100%
Repo Creation	0.0	AUG 27, 2025 1:30 PM	SEP 12, 2025 11:59 PM Late Due Date: OCT 12, 2025 11:59 PM	17	100%
HW01: LinkedList & HashTable	250.0	AUG 29, 2025 11:59 PM	SEP 9, 2025 11:59 PM	16	100%
HW00: Cowsay	100.0	AUG 27, 2025 1:30 PM	SEP 9, 2025 11:59 PM	18	100%

Penn Engineering
University of Pennsylvania



Midterm Logistics





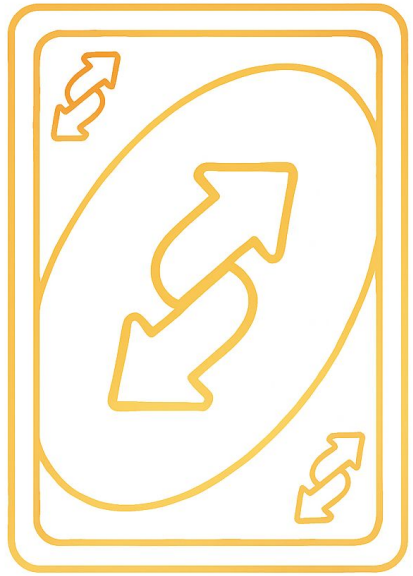
Midterm details

- **Wed, October 22nd 2025**, during lecture, 12:00pm - 1:30pm in AGH 203, our usual room
- You are allowed one double-sided sheet of notes
- Topics are: everything taught up until the exam
  - C concepts, C++ additions, STL, git, System Calls, Locality, Processes, Threads
- https://www.cis.upenn.edu/~tqmcgaha/cis3990/25fa/exams/midterm

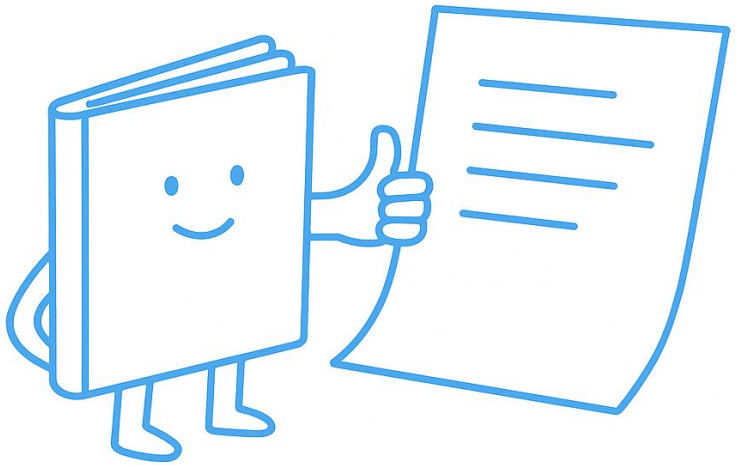
Study resources

- Couse & Recitation slides, Ed questions, HW assignments
- Instructor & Staff OH
- CIT 5950 practice questions & past midterms
  - Note your exam will be a bit harder, CIT 5950 assumes only one semester of prior programming experience. Also, there is significant, but not complete overlap between the 2 courses.

Penn Engineering
University of Pennsylvania

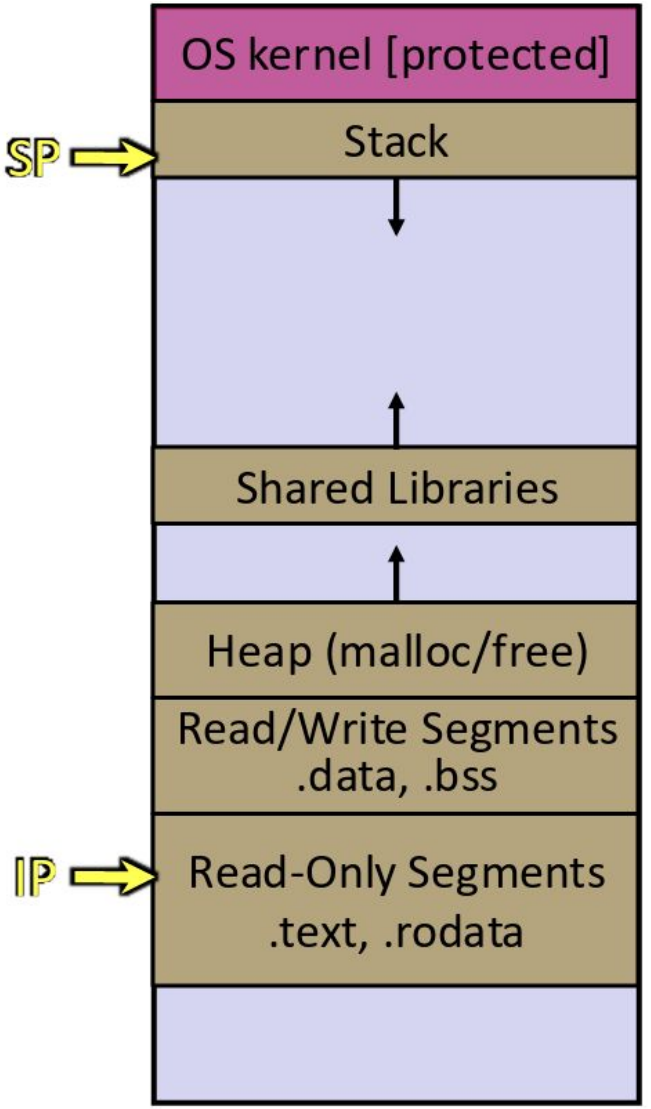


Content review



C stuff 👴

- Almost all legal C is legal C++
- Most relevant C-specific stuff:
  - Pointers
  - Memory layout and allocation - Stack frames, where stuff goes



C stuff 👴- Practice

- Draw the abstract memory diagram for this program, executed until line 13.
  - Where are *x, y, arr, i_msg, o_msg, ans*?
- Where does *ans* point once *my_gen* returns?
- Can I modify the string in *i_msg*? What about *o_msg*?
- What are the bugs and bad practices in the program?

```
1  #include <cstdlib>
2  #include <iostream>
3
4  char* my_gen(int x) {
5    char arr[x + 1];
6
7    for (int i = 0; i < x; i++) {
8      arr[i] = 'a' + rand() % ('z' - 'a' + 1);
9    }
10   arr[x] = '\0';
11
12   std::cout << "My_gen says: " << arr << std::endl;
13
14   return arr;
15 }
16
17 int main() {
18   int y;
19
20   char* i_msg = "Give me a number: ";
21   char o_msg[] = "Here is your string: ";
22
23   std::cout << i_msg;
24   std::cin >> y;
25
26   char* ans = my_gen(y);
27
28   std::cout << o_msg << ans << std::endl;
29
30   return 0;
31 }
32
```

- Pass by reference - the reference is the object itself
- Nicer syntax than pointers
- Prevents unnecessary copies of objects - especially important for function calls and iterations
- Works like *type * const*

```cpp
#include <iostream>

void my_swap(int& a, int& b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int& z = x;

  my_swap(x, y);

  std::cout << "x: " << x << " y: " << y << " z: " << z << std::endl;
  // x: 40 y: 30 z: 40

  z = 50;

  std::cout << "x: " << x << " y: " << y << " z: " << z << std::endl;
  // x: 50 y: 30 z: 50

  return 0;
}
```

Penn Engineering
University of Pennsylvania

Classes, i.e. fancy structs

- RAII - Resource Acquisition Is Initialization
- Rule of 5:
  - Destructor:
    - *~ClassName()*
  - Copy Constructor:
    - *ClassName(const ClassName&)*
  - Copy Assignment Operator:
    - *ClassName& operator=(const ClassName&)*
  - Move Constructor:
    - *ClassName(ClassName&&)*
  - Move Assignment Operator:
    - *ClassName& operator=(ClassName&&)*

©++ - Sample Rule of 5

Penn Engineering
University of Pennsylvania

```
Str::Str() {
  data = new char[11];
  memset(data, '\0', 11);
  capacity = 10;
}

Str::Str(const char* str) {
  data = new char[strlen(str) + 1];
  capacity = strlen(str);
  strcpy(data, str);
}

// Copy constructor
Str::Str(const Str& other) {
  data = new char[other.capacity + 1];
  strcpy(data, other.data);  // deep copy
  capacity = other.capacity;
}

// Move constructor
Str::Str(Str&& other) noexcept {
  data = other.data;
  capacity = other.capacity;
  other.data = nullptr;
}
```

```
Str::~Str() {
  delete[] data;
}

Str& Str::operator=(const Str& other) {
  if (this != &other) {
    delete[] this->data;
    this->data = new char[other.capacity + 1];
    this->capacity = other.capacity;
    memcpy(this->data, other.data, other.capacity + 1);
  }
  return *this;
}

Str& Str::operator=(Str&& other) noexcept {
  if (this != &other) {
    delete[] this->data;
    this->data = other.data;
    this->capacity = other.capacity;
    other.data = nullptr;
    other.capacity = 0;
  }
  return *this;
}
```

C++ - Practice

- Out of calls on lines 15-17 which are allowed?
- What will the compiler complain about regarding lines 19-23?

```
1  #include <iostream>
2
3  void foo(const int& arg) { std::cout << "foo: " << arg << std::endl; }
4
5  void bar(int& arg) { std::cout << "bar: " << arg << std::endl; }
6
7  int main() {
8    int x = 5;
9    int& ref_1 = x;
10   int* ptr_1 = &x;
11   const int& ref_2 = x;
12   const int* ptr_2 = &x;
13   int* const ptr_3 = &x;
14
15   bar(ref_1);
16   bar(ref_2);
17   foo(ref_1);
18
19   ptr_2 = (int*)0xDEADBEEF;
20   ptr_1 = &ref_2;
21   ptr_3 = ptr_3 + 2;
22   *ptr_3 = *ptr_3 + 2;
23   *ptr_2 = *ptr_2 + 1;
24
25   return 0;
26 }
```

C++ - Practice

Penn Engineering
University of Pennsylvania

```
clang++-15 -g3 -gdwarf-4 -Wall -Wpedantic --std=c++2b -O0 const_example.cpp -o const_example
const_example.cpp:16:3: error: no matching function for call to 'bar'
  bar(ref_2);
  ^~~
const_example.cpp:5:6: note: candidate function not viable: 1st argument ('const int') would lose const qualifier
void bar(int& arg) { std::cout << "bar: " << arg << std::endl; }
     ^
const_example.cpp:20:11: error: assigning to 'int *' from 'const int *' discards qualifiers
  ptr_1 = &ref_2;
          ^~~~~
const_example.cpp:21:9: error: cannot assign to variable 'ptr_3' with const-qualified type 'int *const'
  ptr_3 = ptr_3 + 2;
  ~~~~~ ^
const_example.cpp:13:14: note: variable 'ptr_3' declared const here
  int* const ptr_3 = &x;
  ~~~~~~~~~~~^~~~~~~~~~
const_example.cpp:23:10: error: read-only variable is not assignable
  *ptr_2 = *ptr_2 + 1;
  ~~~~~~ ^
4 errors generated.
```

©++ - Practice - Bad_Str

Penn Engineering
University of Pennsylvania

```
5   class Bad_Str {
6    private:
7     char* data;
8     size_t capacity;
9
10   public:
11    // Default constructor
12    Bad_Str() {
13      data = new char[11];
14      memset(data, '\0', 11);
15      capacity = 10;
16    }
17    // Parameterized constructor
18    Bad_Str(const char* str) {
19      data = new char[strlen(str) + 1];
20      capacity = strlen(str);
21      strcpy(data, str);
22    }
23    // Copy constructor
24    Bad_Str(const Bad_Str& other) {
25      data = new char[other.capacity + 1];
26      strcpy(data, other.data);  // deep copy
27      capacity = other.capacity;
28    }
29
30    // Move constructor
31    Bad_Str(Bad_Str&& other) noexcept {
32      data = other.data;
33      capacity = other.capacity;
34    }
```

```
36    // Deconstructor
37    ~Bad_Str() {
38      delete[] data;
39      data = nullptr;
40      capacity = 0;
41    }
42
43    Bad_Str& operator=(const Bad_Str& other) {
44      this->data = new char[other.capacity + 1];
45      this->capacity = other.capacity;
46      memcpy(this->data, other.data, other.capacity + 1);
47      return *this;
48    }
49
50    Bad_Str& operator=(Bad_Str&& other) noexcept {
51      this->data = other.data;
52      this->capacity = other.capacity;
53      delete[] other.data;
54      return *this;
55    }
56
57    // Getters
58    const char * getData() const {return static_cast<const char*>(this->data);}
59    const size_t getCapacity() const {return capacity;}
60  };
61
```

STL

- You should be aware of existing member functions and their operation
- More than comprehensive list here:
- https://en.cppreference.com/w/cpp/container.html#Member_function_table

STL - map pitfall demo

Penn Engineering
University of Pennsylvania

- NEVER search through a map via [] operator!
- RAM goes brrr

Git 

Penn Engineering
University of Pennsylvania

- add
  - Stages modifications for commit
- commit
  - Creates a commit including all the staged changes
- restore
  - Reverts a file to some other state/ unstages changes
- merge
  - Generate a new commit with 2 parens, reconverging the tree
- remote
  - Some “remote” location holding a copy of your repo - likely GitHub
- push
  - Update the remote to match the local
- pull
  - Update the local to match the remote
- branch, checkout
  - Make your tree diverge in a controlled way
- pull requests
  - Reconverge the tree (on GitHub) by making a branch “pull changes” from another branch

Git 🐱🐙 - Practice at home

- learngitbranching.js.org/
- Sample questions for the exam are:
  - What will the tree look like after these commands?
  - What commands should I run to obtain this tree?

System calls 📞

- Defined in POSIX
- C-based “API” for most basic operations on computers
  - File I/O
    - read, write
    - open, close
    - lseek, pipe
  - Process creation and management
    - fork
    - exec
    - waitpid
  - Threads creation and management
    - create, join, etc.
  - Mutex acquisition and management

System calls 📞

- What should we keep in mind when using system level functions?

System calls 📞

- What should we keep in mind when using system level functions?
- When it comes to I/O, what are the performance implications?

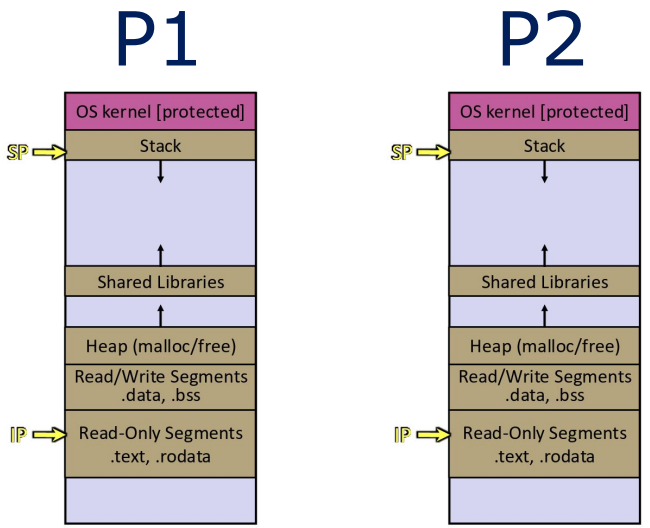
System calls 📞

Penn Engineering
University of Pennsylvania

- What should we keep in mind when using system level functions?
- When it comes to I/O, what are the performance implications?
- How can higher-level functions make our lives harder as programmers?

Processes 

- Every process thinks it has its own memory, registers, everything
- fork() -> duplicate perfectly the calling process, except for the pid. the fork() returns
  - 0 in child process
  - the pid of the child in the parent process
  - -1 on error
- wait() -> tell the kernel to not schedule the calling process until something happens with the process(es) being waited for
  - dying, receiving a signal, etc. depending on the exact wait call



Processes ⏳

Penn Engineering
University of Pennsylvania

- Copy of everything includes copy of file descriptors - whatever the parent had access to, the child does as well
- After fork, processes can independently open and close their fds, without influencing the other processes
- However, if no close and reopen happen, the processes still share the same file cursor - one calling lseek, read, write will influence the other
- Exec asks the system to clear stack, heap, instructions of the calling process, and load up the instructions and data of another process to be run

Processes ⏳ - shared fd demo

Penn Engineering
University of Pennsylvania

```c
int main() {
  int fd = open("two_lines.txt", O_RDONLY);
  char buf[1024];
  memset(buf, '\0', 1024);
  pid_t pid = fork();

  if (pid == 0) {
    read(fd, buf, 10);
    write(2, "Child: ", 7);
    write(2, buf, 1024);
    write(2, "\n", 1);
    close(fd);
  } else {
    int ret_s;
    wait(&ret_s);
    read(fd, buf, 1023);
    write(2, "Parent: ", 8);
    write(2, buf, 1024);
    write(2, "\n", 1);
  }
}
```

```c
int main() {
  char buf[1024];
  memset(buf, '\0', 1024);
  pid_t pid = fork();

  if (pid == 0) {
    int fd = open("two_lines.txt", O_RDONLY);
    read(fd, buf, 10);
    write(2, "Child: ", 7);
    write(2, buf, 1024);
    write(2, "\n", 1);
    close(fd);
  } else {
    int fd = open("two_lines.txt", O_RDONLY);
    int ret_s;
    wait(&ret_s);
    read(fd, buf, 1023);
    write(2, "Parent: ", 8);
    write(2, buf, 1024);
    write(2, "\n", 1);
  }
}
```

Threads 

- Creating more than 1 thread = telling the kernel that, if hardware permits, the current process would like to execute multiple things in parallel
- Threads share memory - we can have data races, non-deterministic outputs, etc. - at the mercy of the scheduler once again
- We can avoid data issues via locks i.e. mutexes

Threads Or Process

Penn Engineering
University of Pennsylvania

For the following, state whether it would be better to use multiple threads or processes:

1. You want to process a big image by calculating the average of all pixels.

2. You have a system of receiving and logging lots of transactions, each action needs data integrity.

3. You have a word processor that constantly checks for spelling mistakes, grammar issues, and syntax errors.

4. You want to enforce security or privacy when performing concurrent transactions.

Threads Or Process

Penn Engineering
University of Pennsylvania

For the following, state whether it would be better to use multiple threads or processes:

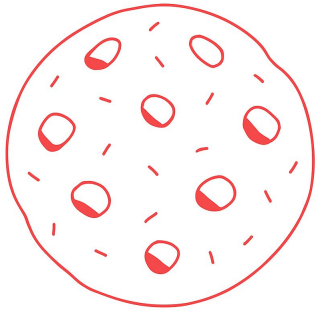
1. You want to process a big image by calculating the average of all pixels. **Threads**

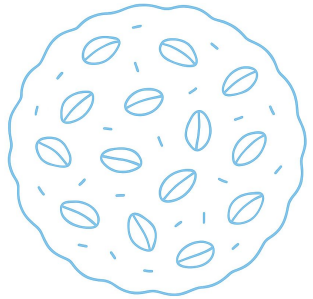
2. You have a system of receiving and logging lots of transactions, each action needs data integrity. **Processes**

3. You have a word processor that constantly checks for spelling mistakes, grammar issues, and syntax errors. **Threads**

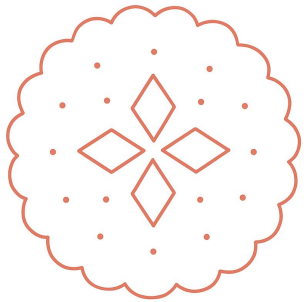
4. You want to enforce security or privacy when performing concurrent actions. **Processes**

Open Q&A





Ask and we shall answer.




That's all Folks!