This week

❖ General programming principles
❖ Biological review

Next week

❖ Dynamic programming
❖ Human Genome paper
❖ Other topics
Outline

- Words of wisdom
- Coding style advice
- How do you validate a program?
- Debugging advice
- Numerical issues
- Performance
- C Programming
  - Random numbers
  - Sorting
  - Pointers
Words of wisdom

❖ "Everything should be made as simple as possible, but no simpler." -- Albert Einstein

❖ KISS principle: “Keep It Simple, Stupid”

❖ From The Zen of Python by Tim Peters:
  • Beautiful is better than ugly
  • Explicit is better than implicit
  • Simple is better than complex
  • Complex is better than complicated
  • Flat is better than nested
  • Sparse is better than dense
  • Readability counts
Coding style advice

❖ Your audience is people as well as the computer
❖ Break large functions into small, simple functions
❖ Break large files into smaller files containing groups of related functions
❖ Use descriptive names for function arguments and variables with larger scopes (e.g. `out_file`, `exons`)
❖ Use short names for iterators, vars of limited scope (e.g. `i`, `j`), and vars that are used many times
How do you validate a program?

❖ Use toy cases with a small amount of data
❖ Figure minimal cases out by hand to verify program
❖ Print intermediate output
❖ Test important functions
Debugging advice

❖ Think!
❖ Print intermediate output
❖ Use toy cases
❖ Use assertions
  • E.g. check probabilities:
    - should always be $\geq 0$ and $\leq 1$
    - often should sum to 1.0
❖ Write slow but sure code to check optimized code
❖ In difficult cases use a debugger, but avoid overuse
❖ valgrind can help find segfaults, memleaks (compile with -g first)
Numerical issues

❖ Consider using log space to avoid overflow and underflow (more on this later in course)

❖ Don’t compare floats with equals (use $\geq$ or $\leq$, NOT $==$)

❖ Beware subtracting large, close numbers

❖ Beware integer casts ($1/2$ is 0, but $1.0/2$ is 0.5)
Performance

❖ Move unnecessary code out of loops

❖ Avoid frequent memory allocation
  • Allocate mem in large blocks (e.g. array of structs, not one at a time)
  • Re-use the same piece of memory

❖ Use a profiler for tough cases (gprof for C; dprofpp for perl)

❖ Avoid slow comparison routines when sorting

❖ Avoid unnecessary optimization!
  - Better to write simple code first and improve speed if necessary
  - Big performance gains often result from changing a few small sections of code (e.g. within loops)
Random numbers (in C)

❖ Don’t use `rand()`
  • use `random()` instead

❖ Use the same seed for reproducible results (e.g. for testing)

❖ Use a different seed (e.g. from the time) for pseudo-random results
#include <stdio.h> /* printf */
#include <stdlib.h> /* qsort */

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *))
{
    int cmp_ints(const void *a, const void *b) {
        int *arg1 = (int *)a;
        int *arg2 = (int *)b;

        if(*arg1 < *arg2) return -1;
        else if(*arg1 == *arg2) return 0;
        return 1;
    }

    int main(int argc, char **argv) {
        int array[] = {-2, 99, 0, -743, 2, 3, 4};
        int i, len = 7;

        qsort(array, len, sizeof(int), cmp_ints);

        for(i = 0; i < len; i++) {
            printf("%d\n", array[i]);
        }

        return 0;
    }
Pointers in C

❖ Pointers are assigned memory addresses (i.e. point to other variables)

❖ The address-of operator (&) obtains the mem addr of a variable

❖ The dereference operator (*) accesses the value stored at the pointed-to mem location

```c
int x;
int *int_ptr; /* declare a pointer to an integer */
x = 10;
int_ptr = &x; /* assign pointer the address of x */

/* dereference ptr obtain value of the pointed-to variable */
printf("The integer value at address %p is %d\n", int_ptr, *int_ptr);

/* assign a new value to the pointed-to variable */
*int_ptr = 12;
printf("x is now %d\n", x);
```
Arrays are pointers to blocks of memory

Array indices are just pointer arithmetic and dereferencing combined:

- `a[12]` is the same as `*(a+12)`
- `&a[3]` is the same as `a+3`
Attributes of pointed-to structures can be derefenced with “arrow notation”:

- \( a->elem \) is equivalent to \( (*a).elem \)

```c
coord.x = 5;
coord.y = 10;

printf("coord.x = %d, coord.y = %d\n", coord.x, coord.y);
coord_ptr = &coord;
coord_ptr->x = 3;
(*coord_ptr).y = 4;

printf("coord.x = %d, coord.y = %d\n", coord.x, coord.y);
```