CS11 Intro C++

FALL 2015-2016

LECTURE 3

C++ Command-Line Arguments

- C++ programs receive command-line arguments in the same way that C programs do

```
int main(int argc, char **argv)
```

- `argc` is the number of arguments specified (including the name of the program itself)
- `argv` is an array of C-style strings containing the arguments themselves
- `argv` always has `argc+1` elements
  - `argv[argc]` is always set to `NULL`
- Example: `./myprog a "b c" d`
  - `argc` is 4
  - `argv` is `{"./myprog", "a", "b c", "d", NULL}`

Aside: `NULL` and `nullptr`

- C++ doesn't officially define the `NULL` symbol
  - `NULL` is defined in `<cstdlib>`
  - Typically defined to be 0, but not required to be…
- Historically, C++ programs would use 0 for `NULL`
- Biggest problem with `NULL` is that it has no type

```
void foo(int i);
void foo(char *psz);
foo(NULL);   // calls foo(int)!
```

  - (Using 0 for `NULL` doesn't solve this problem…)
- C++11 introduces a new `nullptr` value
  - Defined as a pointer-type

Parsing Command-Line Arguments

- Several different approaches for parsing command-line arguments
- The simplest approach:
  1. Verify that there are enough arguments (print an error message if there are not)
  2. Convert numeric arguments using `atoi()` and `atof()` functions from `<cstdlib>`
  3. Verify that arguments are in the proper range (print an error message if not)
- This is a very limited approach
  - `atoi()` and `atof()` don't report when arguments cannot be parsed as numbers

Example Program

- Example: program that takes two arguments

```
// Show usage information for our program
void usage(const char *progname) {
    cout << "usage:  " << progname << " n f"
         << endl;
    cout << "\tn is an integer in the range 1..10"
         << endl;
    cout << "\tf is a nonnegative float" << endl;
}
```

  - It's nice to use the actual name of the program, as it was invoked

Example Program (2)

- Example, cont.

```
// This program shows how to parse arguments.  Whee.
int main(int argc, char **argv) {
    if (argc != 3) {
        usage(argv[0]);
        return 1;  // Indicate there was an issue
    }
    int n = (int) atoi(argv[1]);      // returns long
    float f = (float) atof(argv[2]); // returns double
    if (n < 1 || n > 10 || f < 0) {  // check ranges...
        usage(argv[0]);
        return 1;  // Indicate there was an issue
    }
    ... // Do whatever else the program does
}
```

Advanced Parsing of Command-Line Args

- More advanced programs use `getopt()` function
  - Supports arguments of form "`-x`" or "`-y` *<value>*"
  - Somewhat complicated to use, but very powerful
- `getopt_long()` is also very popular
  - Supports "short" and "long" forms of arg names
  - e.g. "`-f foo.txt`" and "`--filename foo.txt`"
- More effective number-parsing functions:
  - `strtol()` and `strtod()` return both a number, and info on how much of the string was parsed
  - If the entire string wasn't parsed (e.g. "`123hello`"), can report an error indicating the problem

C++ `bool` Type

- C uses `int`s for logical operations
  - 0 is "false," nonzero is "true"
- C++ adds a new `bool` type for Boolean values
  - New keywords: `true` and `false`
  - Comparisons (`>` `<` `==` `!=` …) produce `bool` values
- Use `bool` data-type instead of `int` for flags or other Boolean variables
  - e.g. functions that indicate equality or inequality

C++ bool Type (2)

- `bool` expressions can be converted to `int`
  - `true` → 1, and `false` → 0
- `int`s and pointers can be converted to `bool`
  - nonzero → `true`, and zero → `false`
- Conversion rules allow code written in typical C style to continue to work
- Example C-style code:

```
FILE *pFile = fopen("data.txt", "r");
if (!pFile) { /* Complain */ }
```

  - `pFile` is converted to `bool` type using above rules

Assertions

- Extremely valuable tool in software development

```
#include <cassert>  // assert.h in C
...
assert(condition);
```

- If *condition* is false, the program will halt.
  - Error is shown, with source file and line number of failure
- Use assertions to enforce assumptions and catch bugs in your program
  - Actual data errors, or logical/flow-control errors
  - (Not invalid input or resource-allocation failures, which should be expected by a well-written program.)
- Get in the habit of using them early and often!

Assertions (2)

- `assert()` function is a **macro**
  - Macros rely on the C/C++ preprocessor to transform your code
- In debug mode, `assert(`*`condition`*`)` inserts a test into your program
  - Will likely slow down your program, especially if you have many and/or expensive tests
  - ***Slow and correct is better than fast and wrong…***
- Assertions can be compiled out once you are confident that your program works
  - `g++ -DNDEBUG myprog.cc -o myprog`

Assertion Tips

- Assertions document and enforce assumptions about how the code should work
  - If you expect something should hold at a certain point in your program, enforce it with an assertion
  - Other programmers will see the assertion and understand that the condition is expected to hold
- Don't check separate conditions in one assert call!
  - `assert(index >= 0 && isValid(value));`
  - If this fails, will simply be told that the assertion failed on this line – not which part failed…
  - *Which problem actually happened??*

Assertion Tips (2)

- Use `assert(false)` to check flow-control issues

```
switch (mode) {   // One case for each mode value
case MODE_ONESHOT:
  ...
case MODE_REPEAT:
  ...
default:
  assert(false);  // Should never get here!
}
```

  - Documents and enforces the expectation that every possible value of mode will be handled
- Shouldn't be needed very often, but very useful for ensuring that flow-control works correctly

This Week's Assignment

- Create a more sophisticated version of your Traveling Salesman Problem solver from Lab 2
- Instead of using brute-force to find best solution, use **genetic algorithms** to find a good one
  - Generally converges to a very good solution, very quickly
  - Can't guarantee that it is the best solution

This Week's Assignment (2)

- General approach:
  - Generate an initial population of random genomes
  - Repeat:

    Score each member of the population

    Sort the population from "best" to "worst"

    Use the top N "best" members to generate new members of the population
    - Pick random parents, perform a crossover operation

    Apply random mutations to the population
- Repeat the loop until answer is "good enough"

TSP Genomes

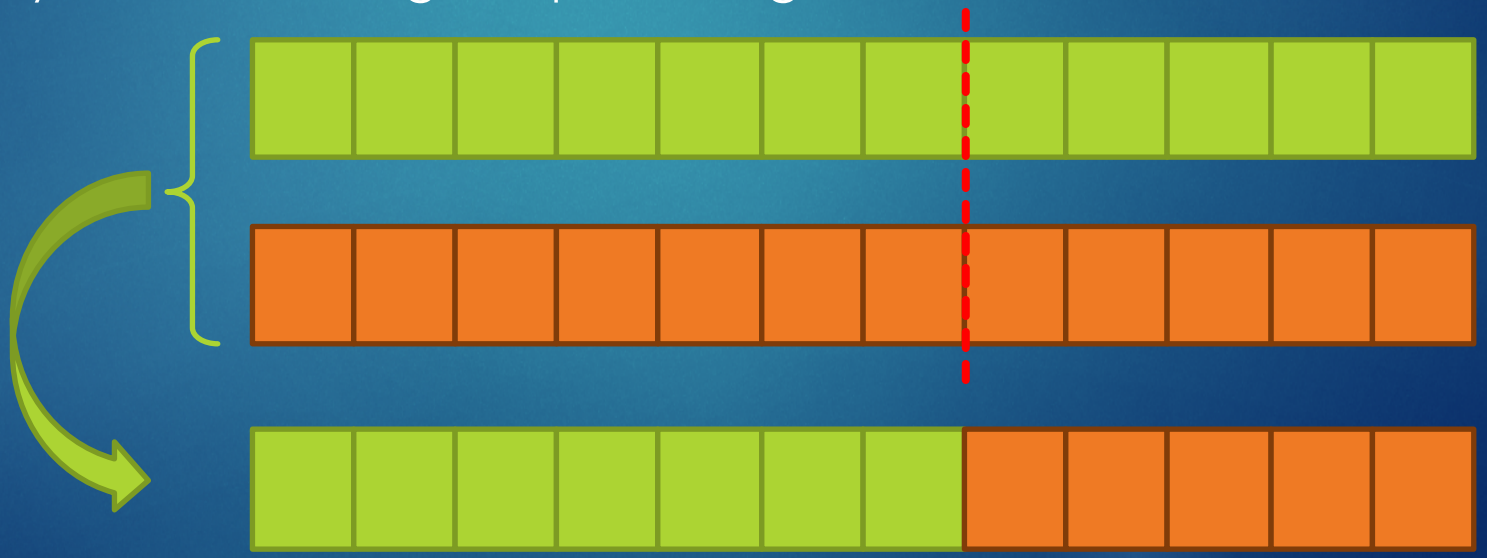
- Use the path order as the "genes" of the genome
  - e.g. for 5 points, genome could be [0 1 2 3 4] to specify the order the points are visited
- Each genome also has a fitness score:
  - The actual distance traveled, when the points are visited in the specified order
- The shorter the path, the "better" or "more fit" the genome is

Sorting Populations

- The STL has a `std::sort()` algorithm
- `std::sort(Iterator begin, Iterator end)`
  - Sorts using the < operator on elements
- `std::sort(Iterator begin, Iterator end, Compare cmp)`
  - Sorts using the comparison function `cmp`
  - `bool cmp(const T&a, const T&b)`
  - `T` is the type of the collection elements
  - Returns `true` if `a < b`, or `false` otherwise
- TSP genomes don't support < operator, but it's easy to write the comparison function

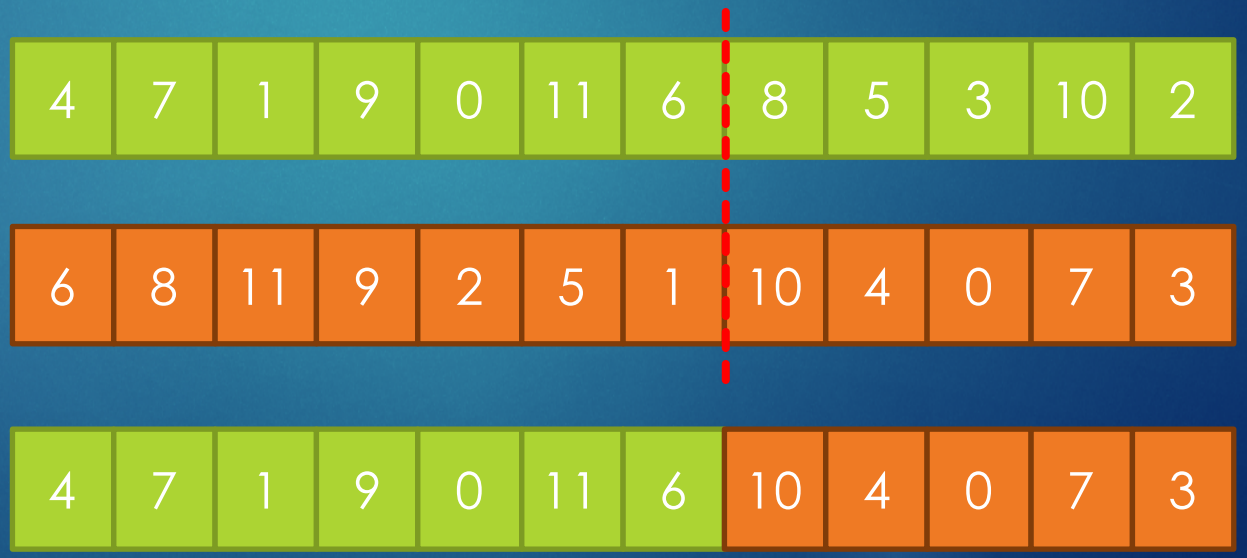
Generating Offspring

- When generating "offspring" for the next generation, two parents are randomly chosen
- "Child" is generated by combining the parents' genomes
  - A crossover point is chosen, and child is generated by concatenating the parents' genes



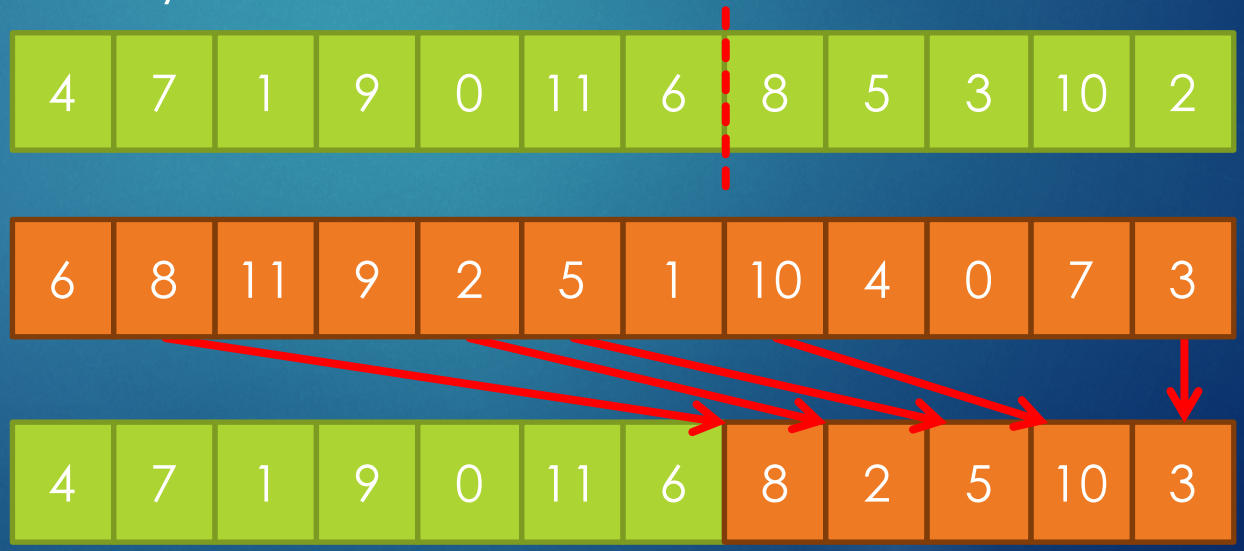
Generating Offspring (2)

- Problem: this procedure will likely generate invalid paths ☹
- Points may appear more than once (0, 4, 7)
- Or, points may be lost (2, 5, 8)



Generating Offspring (3)

- Another approach:
  - Take first part of first parent's genome
  - Then, take numbers from second parent's genome that don't already appear in the first part
  - (take numbers in same order that they appear in second genome)



STL `std::set`

- Can use STL `std::set<T>` collection to track which numbers have been used in the genome
  - `#include <set>`
- `std::pair<Iterator, bool> set<T>::insert(const T &value)`
  - Inserts `value` into the set, if not already present
  - Returns an iterator pointing to the value, and a flag indicating if the value was added or not
- `Iterator set<T>::find(const T&value)`
  - If `value` is in the set, returns an iterator pointing to value in the set
  - If `value` is not in the set, returns `set<T>::end()`

STL `std::set` (2)

- `int set<T>::count(const T &value)`
  - Returns a count of how many times `value` appears in the set
  - For sets, will be 0 or 1

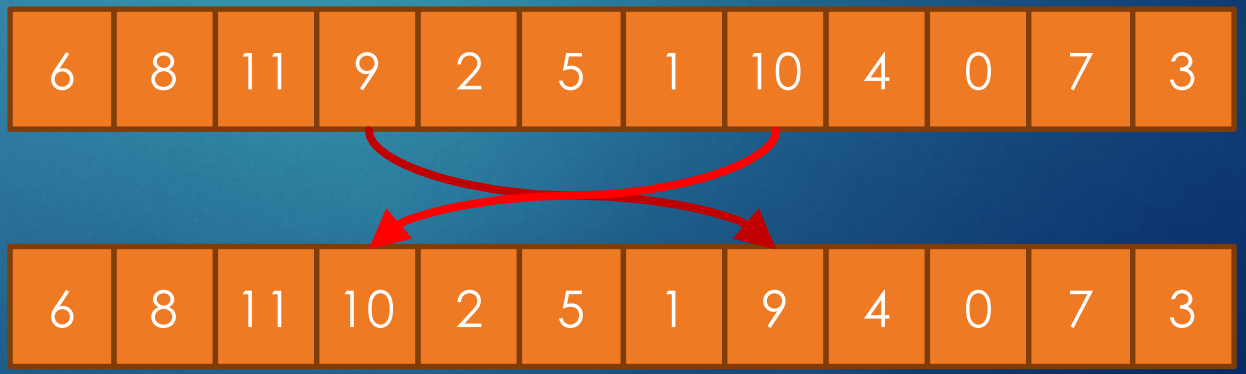
STL std::set (2)

- Example: verify if a genome has no duplicate values

```
bool isGenomeValid(const vector<int> &order) {
    set<int> s;
    for (int i = 0; i < (int) order.size(); i++) {
        if (s.count(i) != 0) {
            cout << "Error:  value " << i
                 << " is repeated!" << endl;
            return false;
        }
        s.add(i);
    }
    // If we got here, no value was duplicated
    return true;
}
```

Mutating Genomes

- Similarly, when mutating genomes, don't want to generate an invalid result
- Implement "mutations" as swaps of two randomly chosen genes



TSP Genetic Algorithm

- This approach converges to an answer extremely quickly
  - Can handle 100+ points
  - Original brute-force approach can't handle > 15-20 points with any speed
- But, you don't know if it's actually the best answer!!! ☺