13. Vector, Command-Line Arguments

CPSC 120: Introduction to Programming
Pratishtha Soni

Agenda

0. Sign-in sheet
    a. Announce: Next class is group worksheet
1. Technical Q&A
2. Scalability
3. Vector
4. Command-Line Arguments

1. Technical Q&A

Technical Q&A

Let’s hear your noted questions about...

- This week’s Lab
- Linux
- Any other technical issues

Reminder: write these questions in your notebook during lab

1. Scalability

Review: Ideal Division of Labor

- **Business Logic:** the human meaning of algorithm data
- Programs
  - **Cannot** understand business logic or design algorithms
  - Can perform tedious, repetitive work flawlessly, quickly, cheaply
- Humans
  - **Can** understand business logic and design algorithms
  - Busy-work is tedious, error-prone, expensive
- Division of Labor Best Practice
  - Humans think about business logic and algorithms
  - Computer programs do repetitive work

Scalability

- **Scalability:** ability of a system to handle increasing workload
- **Scalable program:** can handle increasing amount of INPUT
- Examples
  - Spreadsheet handles hundreds of rows
  - Our Canvas space handles ≈600 users
  - Facebook handles ≈3 billion users
- Major concern for computer science
  - Algorithm efficiency
  - Distributed systems

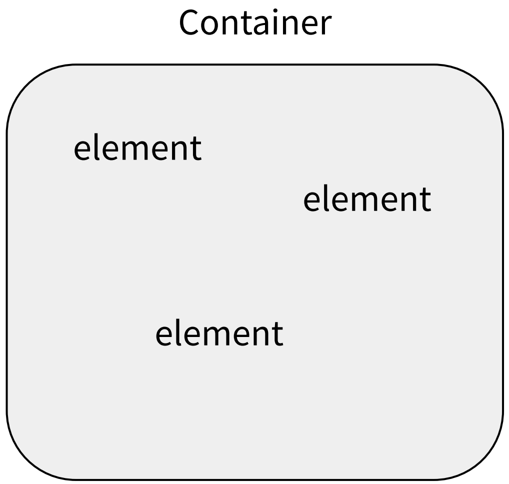
Understand the Problem: Storing Multiple Values

- So far: each data type stores **one** value
  - int, double, bool, std::string, size_t
- Need to store **many** values for scalability
- Declaring individual variables is not scalable

```cpp
std::string student_1_name;
std::string student_2_name;
std::string student_3_name;
// repeat this >600 times?
```

Containers

- **Container:** object that holds other objects
- **Element**: object inside a container
- std::string: text, e.g. “Hello World!”
- std::vector: list
- (more) see *CPSC 131 Data Structures*

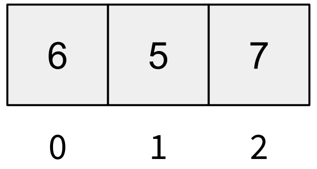


2. Vector

Vector Layout

- **Contiguous:** elements at adjacent memory locations
- **Index:** locations numbered 0, 1, ..., *n*-1

```
std::vector<int> container{6, 5, 7};
```



std::vector is a Template Class

- **Template class:** data type has *template parameter(s)*
- **Template parameter:** fill-in-the-blank

```
template<
    class T
> class vector;
```

```
std::vector<int> container{6, 5, 7};
```

- T is **base type:** data type of each element

Declaring a std::vector

statement:

std::vector<*data-type***>** *identifier* **{** *element ...* **};**

where

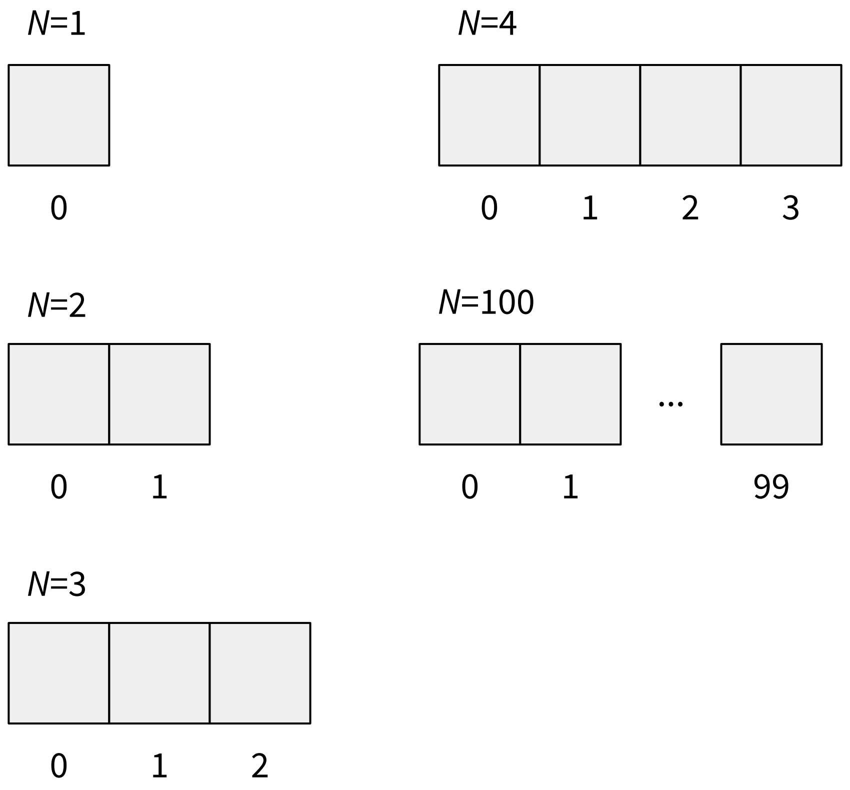
- *data-type* is the type of one element
- *identifier* is variable name
- *element...* are expressions of type *T*

```
std::vector<double> coords{ 1.0, 4.2 };

std::vector<int> phone{2, 7, 8, 1, 7, 1, 2};
```

Valid Indices

- **Index:** position of an element in a vector
- **Indices:** plural of index
- Let *N* = size of vector
- **First** index is 0
- **Last** index is *N* - 1



Runtime Errors

- Recall: **compile errors** are syntax issues that happen at **compile-time**
  - during `$ clang++`
  - source code does not obey syntax pattern
- **Runtime error**: fault in a program that is experienced at **run-time**
  - while program is running
  - during `$ ./a.out`
  - program broke a rule

Exceptions Terminology

- **Exception:** object that represents a runtime error
- **Throw exception:** code creates an exception
- (later) **catch exception:** code receives and handles an exception
- **Uncaught exception:** runtime error that was never handled

Class Member Functions

- Review **class**: compound data type
- Class types we know:
    - std::string
    - std::vector
- Later: create our own class type
- **Member function**: function that operates upon a class object
- Examples:
    - std::cin::good
    - std::cin::clear

Syntax: Member Function Call

expression:

object **.** *function* **(** *argument-expr...* **)**

Semantics:

1. Call *function* on *object* with *argument-expr...* parameters

Examples:

```
int choice{ 0 };
std::cout << “Enter a choice: “;
std::cin >> choice;
if (!std::cin.good()) {
  std::cin.clear();
  std::cout << “Try again: “;
  std::cin >> choice;
}

std::string message{ “hi” };
std::cout << message.size() << “\n”;
```

std::vector::at

- Access an element of a vector
- Member function
- Reference page: std::vector::at , observe
    - pos (index)
    - throws exception for invalid index
    - examples

Identifying a Runtime Error

```
std::vector<int> container{6, 5, 7};
std::cout << "index 0 holds " << container.at(0) << "\n";
std::cout << "index 1 holds " << container.at(1) << "\n";
std::cout << "index 2 holds " << container.at(2) << "\n";
std::cout << "index 3 holds " << container.at(3) << "\n";
```

```
$ ./a.out
index 0 holds 6
index 1 holds 5
index 2 holds 7
terminate called after throwing an
instance of 'std::out_of_range'
  what():  vector::at: __n (which is 3)
>= _Nm (which is 3)
Aborted (core dumped)
```

Identifying a Runtime Error

Message says...

- **Runtime error** (“aborted”)
- **Exception type** (std::out_of_range)
- **Function** (vector::at)
- **Parameter** (__n which is 3)
- strong clues!

```
$ ./a.out
index 0 holds 6
index 1 holds 5
index 2 holds 7
terminate called after throwing an
instance of 'std::out_of_range'
  what():  vector::at: __n (which is 3)
>= _Nm (which is 3)
Aborted (core dumped)
```

std::vector::size

- Member function std::vector::size
- Returns the size of the vector
  - (number of elements)
- Needed when vector is filled at runtime
  - (command-line arguments next)

```
#include <vector>
#include <iostream>

int main()
{
   std::vector<int> nums {1, 3, 5, 7};

   std::cout << "nums contains " << nums.size() << " elements.\n";
}
```

Empty std::vector

- **empty:** contains no elements
- size is zero
- no valid index
  - std::vector::at always throws exception
- Declare with
  - no elements between braces, **or**
  - omit braces entirely
- Classes are always initialized
  - no worry of uninitialized variable

```
// size 0
std::vector<int> scores{};

// size 0
std::vector<double> readings;
```

3. Command Line Arguments

Unix Command Line Arguments

- Recall: in a shell command like

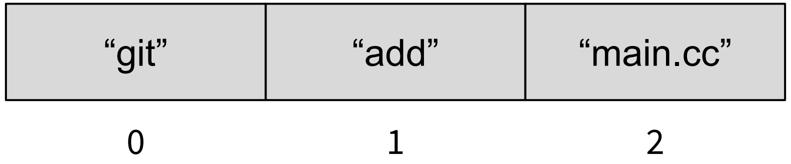
  ```
  $ git add main.cc
  ```

  the command is `git`; there are two **arguments**
    - `add`
    - `main.cc`
- These are INPUT to a Unix program, provided as a list of strings

Argument Array

- argv is an obsolete C-style **array** data structure
- **Command name** comes first (index 0)
- **Arguments** (if any) come next (index 1, 2, ...)

```
$ git add main.cc
```



Initializing a vector of arguments

- Recall main function definition:
  `int main(int argc, char* argv[]) {`
- argc and argv are a C-style array
- Initialize a vector of `std::string` with:

```
int main(int argc, char* argv[]) {
  std::vector<std::string> arguments{argv, argv + argc};
```

- Access strings with `arguments.at(0)`, `arguments.at(1)`, ...

Example: Using Program Arguments

```cpp
#include <iostream>
#include <string>
#include <vector>
int main(int argc, char* argv[]) {
  std::vector<std::string> arguments{argv, argv + argc};
  std::string command{ arguments.at(0) };
  std::string first{ arguments.at(1) };
  std::string second{ arguments.at(2) };

  std::cout << "command: " << command << "\n";
  std::cout << "first argument: " << first << "\n";
  std::cout << "second argument: " << second << "\n";

  return 0;
}
```

```
$ ./a.out peanut butter
command: ./a.out
first argument: peanut
second argument: butter
$ ./a.out 1 2
command: ./a.out
first argument: 1
second argument: 2
$ ./a.out
command: ./a.out
terminate called after throwing an instance of
'std::out_of_range'
what():  vector::_M_range_check: __n (which is 1) >=
this->size() (which is 1)
Aborted (core dumped)
```

Validating Number of Command Line Arguments

- **Problem:** if user provides too few command line arguments, program crashes with `std::out_of_range` error
- **Want:** program handles this gracefully
- **Solution:** when argument count is wrong, print a command error and stop

Review: Input Validation

- **Valid input:** user input
  - exists
  - proper data type
  - proper value
- **Invalid input:** not valid
- **Input validation:**
  - program checks if input is valid
  - when invalid, provides command error and exit code
- Two kinds of invalid input:
  - Extraction failure
  - Range errors

Getting the Number of Arguments

- Recall: we create a variable to hold the arguments:
  `std::vector<std::string> arguments{argv, argv + argc};`
- Get size of the vector with `arguments.size()`
- Recall: includes command name
- `if` statement decides whether size is valid

Putting it Together

1. if statement decides when the number of arguments is wrong
2. Use arguments.size() in the `if` expression
3. Inside the if’s controlled statement,
    a. Use cout to print a command error
    b. return a nonzero exit code

```
if ( arguments.size() != expected-count ) {
  std::cout << “human-readable command error message”;
  return 1;
}
```

Example: Validating Command Line Arg's

```cpp
...
int main(int argc, char* argv[]) {
  std::vector<std::string> arguments{argv, argv + argc};
  if (arguments.size() != 3) {
    std::cout << "error: you must supply two arguments\n";
    return 1;
  }
  std::string command{ arguments.at(0) };
  std::string first{ arguments.at(1) };
  std::string second{ arguments.at(2) };

  std::cout << "command: " << command << "\n";
  std::cout << "first argument: " << first << "\n";
  std::cout << "second argument: " << second << "\n";

  return 0;
}
```

```
$ ./a.out peanut butter
command: ./a.out
first argument: peanut
second argument: butter
$ ./a.out peanut
error: you must supply two arguments
$ ./a.out
error: you must supply two arguments
$ ./a.out peanut butter sandwich
error: you must supply two arguments
```

Look Before You Leap

- **Look Before You Leap:** confirm safety **before** doing risky thing
  - confirming after doesn't help
- Do you...
  - Look then cross? or
  - Cross then look?
- Validate arguments size **before** accessing elements

Pitfall: Validating After Access

```
...
int main(int argc, char* argv[]) {
  std::vector<std::string> arguments{argv, argv + argc};
  std::string command{ arguments.at(0) };
  std::string first{ arguments.at(1) };
  std::string second{ arguments.at(2) };
  if (arguments.size() != 3) {
    std::cout << "error: you must supply two arguments\n";
    return 1;
  }

  std::cout << "command: " << command << "\n";
  std::cout << "first argument: " << first << "\n";
  std::cout << "second argument: " << second << "\n";

  return 0;
}
```

```
$ ./a.out
command: ./a.out
terminate called after throwing an instance of
'std::out_of_range'
what():  vector::_M_range_check: __n (which is 1) >=
this->size() (which is 1)
Aborted (core dumped)
```

Need:

```
$ ./a.out
error: you must supply two arguments
```

Review: Validating Command Line Arg’s

```
...
int main(int argc, char* argv[]) {
  std::vector<std::string> arguments{argv, argv + argc};
  if (arguments.size() != 3) {
    std::cout << "error: you must supply two arguments\n";
    return 1;
  }
  std::string command{ arguments.at(0) };
  std::string first{ arguments.at(1) };
  std::string second{ arguments.at(2) };

  std::cout << "command: " << command << "\n";
  std::cout << "first argument: " << first << "\n";
  std::cout << "second argument: " << second << "\n";

  return 0;
}
```

```
$ ./a.out peanut butter
command: ./a.out
first argument: peanut
second argument: butter
$ ./a.out peanut
error: you must supply two arguments
$ ./a.out
error: you must supply two arguments
$ ./a.out peanut butter sandwich
error: you must supply two arguments
```