ECE2800J

Programming and Introductory Data Structures

Enum, Passing Arguments to Program, I/O

Learning Objectives:

Know when and how to use enum type

Know how to write more general programs that can take arguments

Understand I/O streams

Categorizing Data

Introducing enums

- In addition to single constants, we may need to categorize data.

- For example, there are four different suits in cards:
  - Clubs 
  - Diamonds 
  - Hearts 
  - Spades 

- You could encode each of these as a separate integer like:

```
const int CLUBS = 0;
const int DIAMONDS = 1;
// and so on…
```

Categorizing Data

Introducing enums

```
const int CLUBS = 0;
const int DIAMONDS = 1;
```

- Unfortunately, encoding information this way is not very convenient.

- For example, consider the predicate `isRed()`

```
bool isRed(int suit);
 // REQUIRES: suit is one of Clubs,
 //           Diamonds, Hearts,
 //           or Spades
 // EFFECTS:  returns true if the color
 //           of this suit is red.
```

Categorizing Data

Introducing enums

```
const int CLUBS = 0;
const int DIAMONDS = 1;

bool isRed(int suit);
// REQUIRES: suit is one of Clubs,
//           Diamonds, Hearts, or Spades
// EFFECTS:  returns true if the color
//           of this suit is red.
```

- This is annoying, since we need this REQUIRES clause; not all integers encode a suit.
- There is a better way: the **enumeration** (or **enum**) type.

Categorizing Data

enums

- You can define an enumeration type as follows:

```
enum Suit_t {CLUBS, DIAMONDS,
             HEARTS, SPADES};
```

- To define variables of this type you say:

```
Suit_t suit;
```

- You can initialize them as:

```
Suit_t suit = DIAMONDS;
```

- Once you have such an enum type defined, you can use it as an argument, just like anything else.
- Enums are passed by-value, and can be assigned.

Categorizing Data

enums

- With enum, the specification for the function `isRed()` can be simplified by removing the REQUIRES clause.

```
bool isRed(Suit_t s);
// EFFECTS:  returns true if the color
//           of this suit is red.
```

Categorizing Data

enums

```
bool isRed(Suit_t s) {
  switch (s) {
      case DIAMONDS:
      case HEARTS:
            return true;
            break;
      case CLUBS:
      case SPADES:
            return false;
            break;
      default:
            assert(0);
            break;
  }
}
```

Categorizing Data

enums

- If you write

```
enum Suit_t {CLUBS, DIAMONDS,
             HEARTS, SPADES};
```

  then numerically

```
CLUBS = 0, DIAMONDS = 1,
HEARTS = 2, SPADES = 3
```

- Using this fact, it will sometimes make life easier

```
Suit_t s = CLUBS;
const string suitname[] = {"clubs",
    "diamonds", "hearts", "spades"};
cout << "suit s is " << suitname[s];
```

References

- enum
  - C++ Primer, 4th Edition, Chapter 2.7

Question1: Can I do this?

- enum COLOR1{red, blue, yellow};
- enum COLOR2{pink, purple, yellow};
- (yellow appears in two different enum?)

Question2: Can I do this?

- enum COLOR1{
  red = -1,
  blue = 1,
  yellow = -2
  };

Question3

- enum COLOR1{
  red=-1,
  blue=1,
  yellow
  };

- What's the value of yellow?

Passing Arguments to Program

Introduction

- So far, we have considered programs that take no arguments
  - You run your program like: ./program

- However, programs can take arguments.

- For example, many Linux commands are programs and they take arguments!
  - **`diff file1 file2`**
  - **`rm file`**
  - …

Passing Arguments to Program

Introduction

```
diff file1 file2
```

- The first word, `diff`, is the **name** of the program to run.
- The second and third words are **arguments** to the `diff` program.
- These arguments are passed to `diff` for its consideration, like arguments are passed to functions.
- The operating system collects arguments and passes them to the program it executes.

Passing Arguments to Program

- Arguments are passed to the program through main() function.

- We need to change the argument list of main():
  - Old: `int main()`
  - New: `int main(int argc, char *argv[])`

Passing Arguments to Program

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

- Each argument is just a sequence of characters.
- All the arguments (including program name) form an array of C-strings.
- `int argc:` the number of strings in the array
  - E.g., `diff file1 file2: argc = 3`
  - The name `argc` is by convention and it stands for "argument count".

Passing Arguments to Program

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

- `argv` stores the array of C-strings.
  - Remember, a C-string is itself an array of char and it can be thought of as a pointer to char.
  - Thus, an array of C-strings can be thought of as an array of pointers to char.
  - Thus, `argv` is an array of pointers to char: `char *argv[]`
  - The name `argv` is again by convention and it is short for "argument vector" or "argument values".

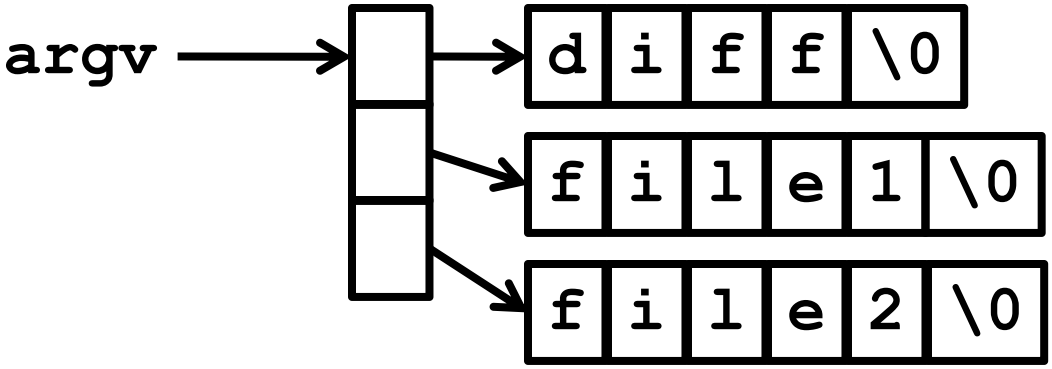
Passing Arguments to Program

argv

```
diff file1 file2
```

```
char *argv[]
```

- Pictorially, this would look like the following in memory:



Note: **argv[0]** is the first string you type to issue the program. It includes the name of the program being executed and optional path (like "./").

Passing Arguments to Program

Example

- Suppose we wanted to write a program that is given a list of integers as its arguments, and prints out the sum of that list.

- Before we can write this program we need a way to convert from C-strings to integers.
- We use predefined "standard library" function called `atoi()`.
- Its specification is

```
int atoi(const char *s);
// EFFECTS: parses s as a number and
//          returns its int value
```

  - Needs `#include <cstdlib>`

Passing Arguments to Program

Example

- The problem we are examining can be solved as:

```
int main (int argc, char *argv[])
{
  int sum = 0;
  for (int i = 1; i < argc; i++) {
      sum += atoi(argv[i]);
  }
  cout << "sum is " << sum;
  return 0;
}
```

Passing Arguments to Program

Example

```
int main (int argc, char *argv[]) {
        int sum = 0;
        for (int i = 1; i < argc; i++) {
                sum += atoi(argv[i]);
        }
        cout << "sum is " << sum;
        return 0;
}
```

- Finally, we save it to `sumIt.cpp`, compile, and run it:

```
$ g++ -o sumIt sumIt.cpp
$ ./sumIt 3 10 11 12 19
```

For the previous command, select all the correct answers

```
$ ./sumIt 3 10 11 12 19
```

- **A.** argc equals 5.
- **B.** argv contains exactly "3", "10", "11", "12", "19" .
- **C.** argv[0] equals "sumIt".
- **D.** The command outputs "sum is 55".

```
int main (int argc, char *argv[]) {
        int sum = 0;
        for (int i = 1; i < argc; i++) {
                sum += atoi(argv[i]);
        }
        cout << "sum is " << sum;
        return 0;
}
```

Passing Arguments to Program

Exercise

- Write a program that is given a list of **floats** as its arguments, and prints out the sum of that list.

References

- Command-Line Arguments
  - Absolute C++, 4th Edition, Page 373

Outline

- Know when and how to use enum type
- Know how to write more general programs that can take arguments
- **Understand I/O streams**

I/O

- **I/O Streams**
  - **Overview**
  - **Output Stream cout**
  - Input Stream cin
  - File Stream
  - String Stream

Input/Output

Streams

- A popular model for how input and output is done in computer systems is centered around the notion of a **stream**.
- A stream is just a sequence of data with functions to put data into one end, and take them out of the other.

```
cin >> a;
```

Input/Output

Streams

- Typical streams:

```
keyboard   →   program
display    ←   program
file       →   program
file       ←   program
string     →   program
string     ←   program
```

- In C++, streams are **unidirectional**.
- Data is always passed through the stream in one direction.
- If you want to read and write data to the same file or device, you need two streams.

Input/Output

Streams

- In general, there are two kinds of stream data: **characters** and **binary data**.
- Characters are usually used for:
  - Communicating between your program and a keyboard or screen.
  - Reading and writing files.
- In addition to text, files can contain arbitrary **binary** data.
  - It is usually much more efficient than character representation.
  - However, it is hard to understand and debug.

- We'll talk about **character streams** here.

Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream

Output Stream: cout

```
cout << "Hello, world!\n";
```

- Output to screen.
- The << is called the **insertion operator**, and is used to insert things into the output stream.
  - It knows how to **convert** all of the other standard data types to **characters** before inserting them into the stream.

```
int foo = 42;
cout << foo << endl;
```

  - Can be cascaded

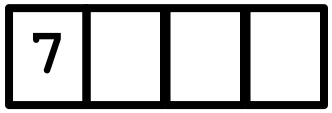
```
cout << foo << " " << bar << endl;
```

Print with Fixed Field Width

```
cout << foo << setw(4) << bar << endl;
```

- Here the `setw()` manipulator sets the width of the **following** number to the specified number of positions and **right-aligns** the number within that field.
- It pads with spaces.

right align 

left align 

- If you want to use `setw()` , you should

```
#include <iomanip>
```

Alternative Output Streams

- You can also use the Linux I/O **redirection** facility to move the output end of the stream from screen to a file:

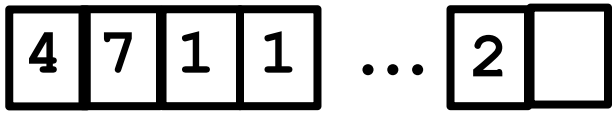
```
$ ./hello > foo
```

- This connects the output end of the `cout` stream to the file "`foo`".

- There is another output stream object defined by the `iostream` library called `cerr`.
- This stream is identical in most respects to the `cout` stream; in particular, its default output is also the screen.
- By convention, programs use the `cerr` stream for **error messages**.

Output: Buffering

- I/O in C++ is **buffered**.
- This means output inserted into an output stream is saved by the underlying operating system (in a region of memory called a **buffer**).



- The content in the buffer is written to the output only when specific actions are taken.

Output: Buffering

- The buffer content is written to the output only when:
  - A newline (e.g., `endl` or '`\n`') is inserted into the stream. E.g.,

    ```
    cout << "ok" << endl;
    ```

  - The buffer is explicitly flushed. E.g.,

    ```
    cout << "ok" << flush;
    ```

  - The buffer becomes full
  - The program decides to read from `cin`
  - The program exits
- Once the buffer content is written to the output, the buffer is **cleaned**
- If some content is not printed out, it may be still in the buffer
- In contrast, output sent to `cerr` is not buffered

References

- **C++ Primer (4th Edision)**, by *Stanley B. Lippman, Josée Lajoie, Barbara E. Moo*, Addison-Wesley Publishing (2005)
  - Chapter 8