WARWICK
THE UNIVERSITY OF WARWICK

C++ command-line arguments

Tom Latham

(based on slides from Matt Williams)

Command-line arguments

- There are two main ways to pass information into a program:
  - Interactively (text or GUI)
    - Step-by-step instructions
    - Good for new users
  - Command-line arguments
    - Reproducible
    - 'Fire and forget'

An example: using g++

- It would be a pain to have to deal with g++ interactively

```
$ g++
Enter name of C++ file: main.cpp
Enter name of output file: myprogram
Enable C++11 [y/n]: y
Enable all warnings [y/n]: y
…
$
```

- Instead we provide the program arguments up-front so we can run the same command over-and-over again with minimal typing

```
$ g++ -Wall -Wextra -std=c++11 -o myprogram main.cpp
```

Arguments to mpags-cipher

- We would like to be able to do the same with our program, e.g.

```
$ ./mpags-cipher -i plain.txt -o cipher.txt -c caesar -k 17 --encrypt
```

- But how do we get the information that the user supplies on the command-line into variables within our program?

- The operating system splits the command line by whitespace and passes it to the program as a list of strings:

```
{"./mpags-cipher", "-i", "plain.txt", "-o", "cipher.txt", "-c", "caesar", "-k", "17", "--encrypt"}
```

- The values are passed to the **main()** function of our program

Reading arguments in C++

- Due to backward compatibility with C, the way that these appear in **main()** are as two function arguments:
  - **argc** is an integer - the number of arguments
  - **argv** is a C-style array of C-style strings - the arguments themselves
- These are rather fiddly to work with, so it is best to immediately convert them into a more easily usable form, a std::vector of std::string objects:

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

- We can then loop over and/or access the individual arguments as with any std::vector

Exercise 6: reading arguments in C++

- *Edit your main function to print out each argument that was passed to the program*
- You'll need to use the code on the previous slide and add a 'for' loop
- Try running your program with different numbers of arguments and make sure it adapts as you would expect

Terminology

- Useful to distinguish between:
  - **argument**
    - These are non-optional parts which are fundamental to the program. e.g. the list of .cpp files passed to g++
  - **option**
    - An optional argument, usually marked by --**output=foo** or **-o foo**
  - **flag**
    - Like an option but without the second part. Changes some behaviour of the program. e.g. **-Wall**

Exercise 7: printing a help message

- A common command-line flag is **-h** or --**help,** which makes the program print some information about how to use the program, e.g.

```
$ g++ --help
Usage: g++ [options] file...
Options:
  -pass-exit-codes   Exit with highest error code from a phase
  --help             Display this information
  --target-help      Display target specific command line options
...
```

- *Edit your program to check for the presence of either of those options (-h or --help) and print some help text*

Handling options

- We can now handle arguments and flags but options are different in that they span more than one entry in the list

```
{"./mpags-cipher", "-i", "plain.txt", "-o", "cipher.txt", "-c", "caesar", "-k", "17", "--encrypt"}
```

- In the above example, in order to determine the name of the output file name, one needs to check for the presence of **-o** and, if found, use the value of the next element to obtain the output file name

- The parsing of the rest of the arguments must then continue from the argument after that, i.e. two after the **-o**

Exercise 8: handle all the options

- *Edit your program to handle* **-h**, **--help**, **--version**, **-i input_file** *and* **-o output_file**
- *Print the appropriate output or, for the files, store the name of the file supplied in a variable and print it out*
- All arguments should be optional and available in any order
- The program should also print appropriate messages and exit if there was a problem parsing the arguments

Using a library for the job

- You've had to write all the code to do the checking manually
- Once the program gets more complicated you may want to automate it
- Most software will use a library for doing this.
  A common one for C++ is provided by Boost as boost::program_options
- Other languages have their own such as Python's argparse

boost::program_options

```cpp
#include <boost/program_options.hpp>
namespace po = boost::program_options;

int main(int argc, char* argv[])
{
  std::string input_file;

  po::options_description desc("Allowed options");
  desc.add_options()
    ("help", "produce help message")
    ("i", po::value(&input_file), "Name of input file");

  po::variables_map vm;
  po::store(po::parse_command_line(argc, argv, desc), vm);
  po::notify(vm);

  if (vm.count("help")) {
    std::cout << desc << "\n";
    return 0;
  }
  ...
}
```