C++ Basics review

CSC 240 - Data structures with C++ - Lyon College, FA24

Marcus Birkenkrahe

July 18, 2024

Overview

Fast paced C++ review and overview including:

1. Structure of C++ programs
2. "Hello world" sample program
3. `main` function, `#include` and `namespace`
4. Declaring and initializing variables
5. Conditional control statements
6. Functions
7. Printing format specifiers
8. Comments
9. Debugging with `gdb` and pythontutor.
10. Simple Exercises for class and home assignments

Just enough to be able to follow the rest of the course!
You should code along for a large part of this lecture!

The Structure of a Basic C++ Program

- What is required to create and run a C++ program?

  1. An editor to create the source code (like Emacs).
  2. A compiler to turn the source code into executable code (like GCC).
  3. A linker to link libraries (like `iostream`) to the source code file.
  4. A shell (or an IDE with a shell) to run the executable (like `bash`).

- What must every C++ program have?

  A `main` function. Optional: Preprocessor directives (like `#include`).

A first C++ program in six incarnations

For this section, fire up Emacs, open a file (`C-x C-f first.org RET)` and code along. To create a C++ code block, enter `<s TAB C++`, to run it, `C-c C-c`. What happens when you run a code block?[1]

1. Example: "Hello world" program - can you explain every line?

```
#include <cstdio>

int main() {

  printf("Hello, world!");

  return 0;
}
```

```
Hello, world!
```

[1]The source code is completed using the header arguments (if any) to create a source file, which is handed over to the compiler. If this file is syntactically correct, it will compile, and then run. Emacs does all of this in the background. Since C++ is a compiled, not an interpreted language (like Python or R), there is no continuous session.

2. A good habit: Provide ample comments (especially at the beginning)

```
/* --------------------------------------------*/
/* Print a string to a standard output device  */
/* --------------------------------------------*/
// include standard input/output library
#include <cstdio>
// main program
int main() {  // return integer, take no argument (void)
  // print string
  printf("Hello, world!");
  // return 0 if successful
  return 0;
} // end of main program
```

```
Hello, world!
```

3. Here is another version: What's different?

```
/* --------------------------------------------*/
/* Print a string to a standard output device  */
/* Use character array and formatted printing  */
/* --------------------------------------------*/
#include <cstdio>
// main program
int main() {  // return integer, take no argument (void)
  // declare and initialize character array
  char greeting[] = {"Hello, world!"};
  // print string
  printf("%s\n", greeting);
  // return 0 if successful
  return 0;
} // end of main program
```

```
Hello, world!
```

4. And another version: What's different?

```
/* --------------------------------------------*/
/* Print a string to a standard output device  */
/* Use iostream and stream extraction cout <<  */
/* --------------------------------------------*/
#include <iostream>
// main program
int main() {  // return integer, take no argument (void)
  // print string
  std::cout << "Hello, world!" << std::endl; // << "\n"; would also work
  // return 0 if successful
  return 0;
} // end of main program
```

```
Hello, world!
```

5. With the magic of Emacs + Org-mode, one could almost believe that C++ was Python or R - what's different?

```
// print string to standard output device
cout << "Hello, world!" << endl;
```

```
Hello, world!
```

6. Literate programming: The last code block can be "tangled" into source code:

```
cat hw.cpp
```

```
#include <iostream>

using namespace std;




int main() {
// print string to standard output device
cout << "Hello, world!" << endl;
return 0;
}
```

7. The source code file can be compiled and run on the command line (or in a shell in this Org-mode file):

   ```
   g++ -o hello hw.cpp
   ./hello
   ```

   ```
   Hello, world!
   ```

`main`

- All C++ programs have a single entry point called the `main` function.
- Functions are blocks of code that can take input and return results:

  ```
  void hello() { // function declaration and definition
    cout << "hello" << endl;
  }
  main() { // program entry point
    hello();  // function call
  }
  ```

  ```
  hello
  ```

- Look at the tangled file:

  ```
  cat hw2.cpp
  ```

  ```
  #include <iostream>

  using namespace std;




  void hello() { // function definition
    cout << "hello" << endl;
  }
  main() { // program entry point
    hello();  // function call
  }
  ```

- When you define your own functions (or classes and their member functions), you need to declare them before `main` (you can define them later).
- In this code, `hello` is declared as a prototype, and defined later:

  ```
  void hello(); // function declaration (prototype)

  int main() { // program entry point
    hello();  // function call
    return 0;
  }

  void hello() { // function definition
    cout << "hello" << endl;
  }
  ```

  ```
  hello
  ```

#include

- Most programming languages incorporate library functions. Libraries are often sizeable and must be installed, sometimes compiled from source, and they're linked to the language version used.
- Python, Go and Java have `import`. Here's a Python example:

  ```
  import numpy # import library (install with ‘pip‘)
  [print(_) for _ in globals()]
  ```

  ```
  __name__
  __doc__
  __package__
  __loader__
  __spec__
  __annotations__
  __builtins__
  codecs
  os
  ```

```
__pyfile
__code
__org_babel_python_format_value
__PYTHON_EL_native_completion_setup
f
numpy
```

- R has `library`:

  ```
  library(MASS) # import stats library (install with ‘install.packages‘
  search()
  ```

  ```
  [1] ".GlobalEnv"        "package:MASS"      "ESSR"
   [4] "package:stats"     "package:graphics"  "package:grDevices"
   [7] "package:utils"     "package:datasets"  "package:methods"
  [10] "Autoloads"         "package:base"
  ```

- Rust and C# have `use`, JavaScript, Lua, Perl and PHP have `require`, and C/C++ have `include`.
- In the C++ examples so far, we included `iostream` (for `cout` and `<<`), and `cstdio` (for `printf`).
- The command `#include` is only one of many possible *preprocessor* directives

namespace

- Namespaces prevent naming conflicts. For example, when importing libraries, namespaces are essential for identifying symbols.
- The keyword `cout` to direct output to standard output devices (like a screen) is defined in the `std` namespace ('standard'). Its full name is therefore `std::cout`.
- You can employ a `using` (preprocessor) directive to avoid a lot of typing:

  ```
  using namespace std;
  cout << "Much shorter than..." << endl;
  std::cout << "...this statement." << std::endl;
  ```

```
Much shorter than...
...this statement.
```

- This is the same thing that's going on when using `import` in Python and choosing an alias: `np` allows you to access all members of the `numpy` library, e.g. the `array` function.

  ```
  import numpy as np
  x = np.array([1,2,3,4,5]) # define
  print(x)
  ```

  ```
  [1 2 3 4 5]
  ```

- In Emacs Lisp (the language most of Emacs is written in), `org-version` is a member of two namespaces: it's the name of a variable, and a function of the same name:

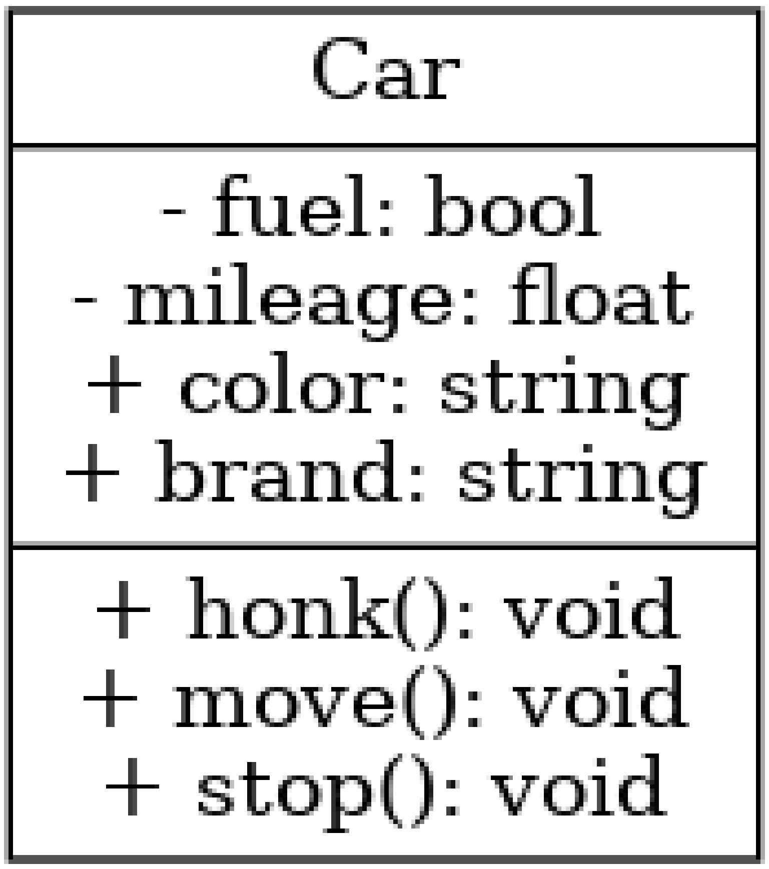
  ```
  (message org-version) ;; extract variable - prints 9.7.7
  (org-version)  ;; run function - prints 9.7.7
  ```

- By default, all symbols you declare go into the global namespace. In C++, you can define your own namespace and place your symbols into it - this affords additional *encapsulation*, an important principle of object-oriented programming.

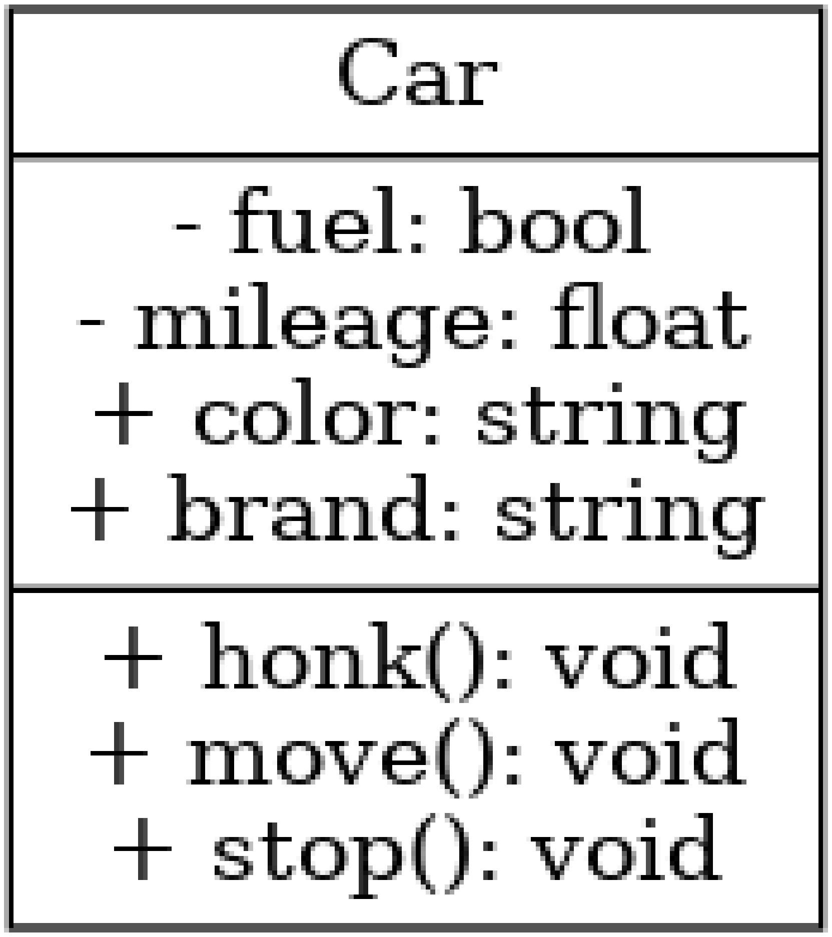
The C++ Type System

- C++ is an object-oriented (OO) programming language. What does that mean?
- Everything in C++ has a state and a behavior, something it is (attribute, feature), and something it can do (method, function).
- Example: A car.
  1. The car's states are moving or standing.
  2. The car has a certain weight, a color, a brand.
  3. The car can move, accelerate, stop, honk.

- In addition, some of the properties of the car can be considered "private" or hidden (e.g. the fuel state - empty or full, or the mileage) while others are "public" or visible (e.g. the color and the brand).
- The Unified Modeling Language (UML) has a diagram ("class diagram") just for the description of objects and their relationships, like this[2]:



[2]To render the `dot` language in Emacs, you need to install graphviz (`sudo apt install graphviz`), and add `(dot . t)` to the other languages already loaded in your `.emacs` configuration file.



- A data type is the collection of behaviors and states that describe an object. In our example, the `car` is a user-defined data type.
- C++ is a *strongly typed language*, which means that each object has a predefined data type.
- Example: `int` represents the built-in integer type.
  1. An `int` object can store whole numbers (state).
  2. An `int` object represents a certain amount of memory (state).
  3. An `int` object supports math operations (behavior).
  4. An `int` object is printed with the `%d` format specifier (state).

- To do anything with objects, you need to name them. Named objects become variables.

Declaring and Initializing Variables

- You declare variables by providing their type, followed by their name, followed by a semicolon:

  ```
  int foo;  // declare integer variable named ‘foo‘
  ```

- When you declare a variable, you initialize it - set its initial state such as setting its value and/or reserving memory for later assignment of a value:

  ```
  // declare
  int foo;
  // define
  foo = 42;
  // declare and define
  int bar = 2;

  // declare and define after evaluating an expression
  int baz = foo * bar;

  // print all variables
  cout << foo << " * " << bar << " = " << baz << " " << endl;
  ```

  ```
  42 * 2 = 84
  ```

- What happens if you divide two integers and the numerator is not a multiple of the denominator?

  ```
  int foo = 42, bar = 4;
  cout << foo/bar << endl; // 42 / 4 = 40/4 + 2/4 = 10 1/2 = 10.5
  ```

  ```
  10
  ```

- Answer:

```
// Widening conversion
int foo = 42, bar = 4;
cout << float(foo)/float(bar) << endl; // 42 / 4 = 40/4 + 2/4 = 10 1/2 = 10.5
```

```
10.5
```

- The other common built-in data types besides `int` are `float` and `bool`.

Conditional Statements

- Conditional statements allow you to make decisions based on Boolean expressions, which evaluate to true or false.
- You can use comparison operators like `==` or `<=` to build up Boolean expressions that represent logical choices: "When the sun shines, the weather is good," can be translated into this C++ code:

```
// premise
bool sun = false; // true = 1 or false = 0 also works
// conclusion
if (sun == true) {
  // good weather: sun is shining
  cout << "The weather is good" << endl;
 } else {
  // bad weather: sun is not shining
  cout << "The weather is bad" << endl;
 }
```

```
The weather is bad
```

- Can you change the program to represent the following statement: "When the sun shines and there's rain, we're getting rainbows."

```
// premise
bool sun = true, rain = true;
// conclusion
if (sun == true and rain == true) { // && would also work instead of ‘and‘
  // good weather: sun is shining
  cout << "We're getting rainbows" << endl;
 } else {
  // bad weather: sun is not shining
  cout << "We're not getting rainbows" << endl;
 }
```

```
We're getting rainbows
```

- If the first `if` condition is not true, the `else` statement is executed. You can also have any number of choices tested in cascading `else if` conditions (the `else` is optional):

```
int x = 0;
if (x > 0) printf("Positive.");  // branch one
 else if (x < 0) printf("Negative."); // branch two
 else printf("Zero."); // default branch
```

```
Zero.
```

Functions

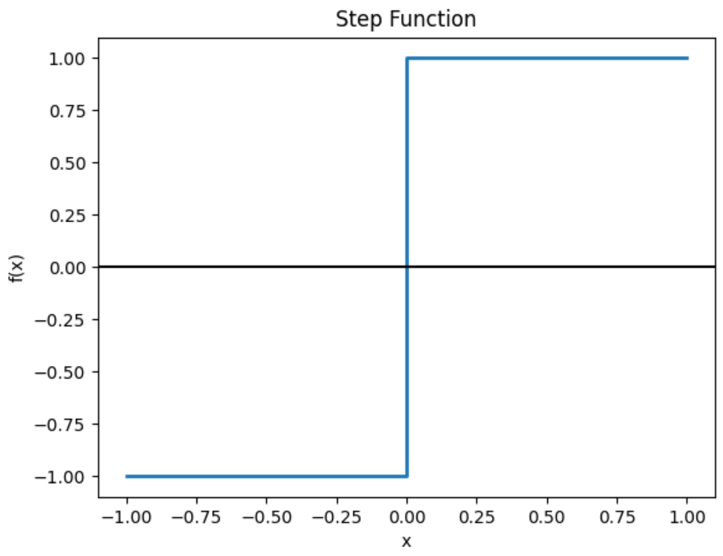
- Functions are code blocks that accept any number of input objects called *parameters* or *arguments*, and that can `return` output objects when called.
- They have a `return` type. If no such type is specified, they are `void` (better to specify that, too).
- Example: A mathematical step Function. What is it? Is it important[3]?

  The step function is -1 for all arguments smaller than zero, 1 for all arguments greater than zero, and zero for a zero argument.

- It's useful to plot functions in 2D if you can:

[3]Step functions are important e.g. as activation functions for neural nets, in signal processing and digital circuits, and wherever abrupt changes are being modeled.

```
## Plot a step function - mathematically:
## \forall x \in [-1,0): f(x) = -1, \forall x \in (0,1]: f(x) = 1
## ------------------------------------------------
# import graphics library
import matplotlib.pyplot as plt
# x and f(x) = y
x = [-1,0,0,1]
y = [-1,-1,1,1]
# plot f(x)
plt.clf() # clear graphics
#plt.plot(x,y,marker='o',linestyle="")  # shows the four points
plt.plot(x,y,linewidth=2) # draws a line between them
plt.axhline(0,color="black")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Step Function")
plt.savefig("../img/step.png")
```



- Before writing the code, let's understand what we're after:
  1. the function should take one integer argument
  2. the function should return one of: -1, 0, 1

3. the return value depends on the argument

```
// define function
int step(int x) {
    // compute result for x < 0
    // compute result for x = 0
    // compute result for x > 0
    // return result
}
```

- We call the function from `main`

```
// define step function
int step(int x) {
  int y = 0; // initialize y
  if (x < 0) { // compute y for negative x
    y = -1;
  } else if (x > 0) { // compute y for positive x
    y = 1;
  }
  return y;
}
// main function
int main() {
  // call function for negative integer
  cout << step(-10) << endl;
  // call function for zero
  cout << step(0) << endl;
  // call function for positive integer
  cout << step(100) << endl;
  return 0;
}
```

```
-1
0
1
```

Print format specifiers

- `cout` with `<<` is pretty comfortable but `printf` is a lot more flexible.

- The first argument of `printf` is always a *format string*, like `%m.pf` `float` values with `m` spacees and a precision of `p`
- Example: To print 3.1459 righ-aligned on 10 places:

  ```
  printf("%10.4f\n", 3.1459);
  printf("|----|----|");
  ```

  ```
      3.1459
  |----|----|
  ```

-
- People have strong opinions when it comes to teaching C++ I/O:
  1. `cstdio::printf` goes back to C and has a lot more flexibility but is also more error prone.
  2. `iostream::cout` is part of the C++ standard library but involves a lot of complicated concepts (stream buffers, `<<` operator, `flush` into a *destructor* method, `setf` etc.).

Comments (and Pseudocode)

- Comments (like pseudocode) are non-standard and subjective.
- You can't overdo it with comments when learning a language
- Professional commenting is different (for code sharing)
- When you use literate programming techniques and apps (like Emacs + Org-mode, or Jupyter, or noweb), you don't need any comments at all.
- At the start, you should write pseudocode for every new algorithm, and your pseudocode should turn into comments
- The most important comment is the multi-line description of your program at the top.
- Example (Smith, 2023): write pseudocode for the following problem.

This program will request a student's score from the user. If the score is above 60 then a "passed" message will be delivered; if not, a "failed" message will be delivered.

- Sample solution:

  1. First attempt - what are the pros and cons?

     ```
     // load I/O library

     // get user input

     // compute result

     // print result
     ```

     - Pros: task completely covered (start-end), syntax-free
     - Cons: lacks necessary detail, especially for algorithm

  2. Second attempt - what are the pros and cons?

     ```
     /* ------------------------------------------------------------ */
     /* Get score from user and return "passed" or "failed" message. */
     /* Sample input: 65                                             */
     /* Sample output: "passed"                                      */
     /* ------------------------------------------------------------ */

     // include I/O library

     // declare variables: float score

     // ask for score between 0 and 100

     // get user input from keyboard

     // compute result

     // stream result to screen
     ```

     - Pros: program header useful (later), includes input/output, variable detail; good for planning the whole program
     - Cons: lacks focus on algorithm (still no detail here); detail obscures logic of the solution

3. Third attempt:

```
// If grade is greater or equal to 60
   // print "passed"

// else
   // print "failed"
```

   - Pros: clarifies algorithm without getting bogged down in syntax (but suggests enough syntax to remember/structure the code)
   - Cons: Leaves out "standard stuff" (like I/O), problematic for beginners only.

4. Implementation with comments:

```
/* ---------------------------------------------------------- */
/* Get score from user and return "passed" or "failed" message. */
/* Sample input: 65                                            */
/* Sample output: "passed"                                     */
/* ---------------------------------------------------------- */
#include <iostream> // include I/O library

int main() {
  // declare variables: float score
  float score;
  // ask for score between 0 and 100
  cout << "Enter score between 0 and 100: ";
  // get user input from keyboard
  cin >> score;
  // compute result
  cout.precision(3); // set output precision
  // If grade is greater or equal to 60
  if (score >= 60) {
 cout << "\n" << score << ": passed" << endl;  // print "passed"
  } else {
 cout << "\n" << score << ": failed" << endl;  // print "failed"
  }
  return 0;
}
```

```
Enter score between 0 and 100:
65: passed
```

Input file:

```
echo "65" > ../data/score
cat ../data/score

65
```

Debugging

- Let's update the `step` function with `prinf` commands for output, and then step through it with the GNU debugger, `gdb`.
- You have to install `gdb` in terminal (`M-x shell`) with the following super-user command (after an update):

  ```
  sudo apt update -y
  sudo apt install gdb -y
  ```

- Now `gdb --version` should work:

  ```
  aletheia@pop-os:~/GitHub$ gdb --version
  GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
  Copyright (C) 2022 Free Software Foundation, Inc.
  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
  This is free software: you are free to change and redistribute it.
  There is NO WARRANTY, to the extent permitted by law.
  ```

- The code now includes `<cstdio>` for `printf` and the declaration and definition of a few variables in the `main` program:

  ```
  int step(int x) {
    int result = 0;
    if (x < 0) {
      result = -1;
    } else if (x > 0) {
      result = 1;
    }
    return result;
  }
  int main() {
  ```

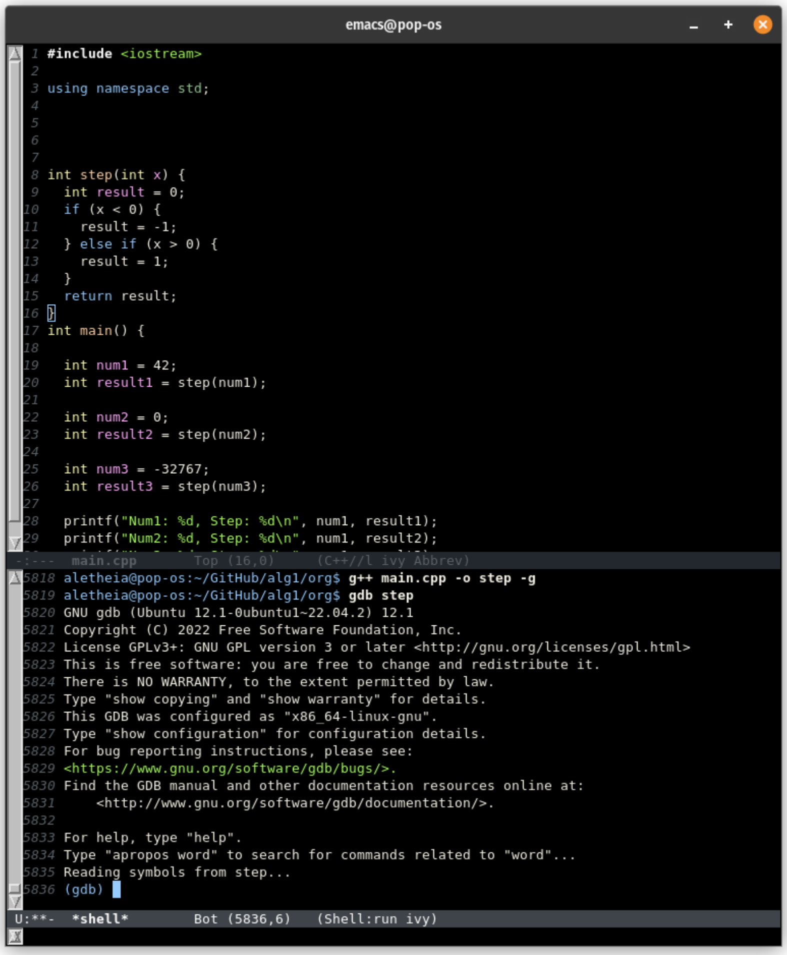
```
  int num1 = 42;
  int result1 = step(num1);

  int num2 = 0;
  int result2 = step(num2);

  int num3 = -32767;
  int result3 = step(num3);

  printf("Num1: %d, Step: %d\n", num1, result1);
  printf("Num2: %d, Step: %d\n", num1, result2);
  printf("Num3: %d, Step: %d\n", num1, result3);

  return 0;
}
```

```
Num1: 42, Step: 1
Num2: 42, Step: 0
Num3: 42, Step: -1
```

- Before you can step into `gdb`, tangle (`C-u C-c C-v t`) and compile your source code using this command:

  ```
  g++ main.cpp -o step -g
  ```

- The `-g` flag generates debugging information for `gdb`.
- Now split the Emacs screen into two windows with `C-x 2`. Keep the source code of `main.cpp` in the upper window while you step into the debugger in the lower window. Switch on line number mode with `M-x global-linum-mode` (toggles):



```
emacs@pop-os
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5
 6
 7
 8 int step(int x) {
 9   int result = 0;
10   if (x < 0) {
11     result = -1;
12   } else if (x > 0) {
13     result = 1;
14   }
15   return result;
16 }
17 int main() {
18
19   int num1 = 42;
20   int result1 = step(num1);
21
22   int num2 = 0;
23   int result2 = step(num2);
24
25   int num3 = -32767;
26   int result3 = step(num3);
27
28   printf("Num1: %d, Step: %d\n", num1, result1);
29   printf("Num2: %d, Step: %d\n", num1, result2);
-:---  main.cpp       Top (16,0)     (C++//l ivy Abbrev)
aletheia@pop-os:~/GitHub/alg1/org$ g++ main.cpp -o step -g
aletheia@pop-os:~/GitHub/alg1/org$ gdb step
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from step...
(gdb)
U:**-  *shell*        Bot (5836,6)    (Shell:run ivy)
```

- Commands:

```
$ gdb step      # step into debugger
(gdb) break main # set breakpoint at start of main
(gdb) break main.cpp:17 # set breakpoint at line 17 of main.cpp
(gdb) run  # run program until next breakpoint
(gdb) step # step into function
(gdb) next # execute next command but do not step into function
(gdb) finish # step back out of a function call
(gdb) continue # continue to next breakpoint or until finish
(gdb) info local # current value of variables
(gdb) help  # gdb help system with lots more information
(gdb) quit  # leave debugger
```

- Switch global line number mode off again (`M-x global-linum-mode`).

Exercises

1. Fix the syntax of the following program.

   ```
   #include <iostream>;
   using namespace std

   main() { // begin main function

     // print greeting followed by new line
     cout << "Hello there!" << endl;

     return 0;
   } // end main function
   ```

2. The code is correct but it won't compile with `C-c C-c` as it should. Fix the code. Tip: tangle the chunk to see what's going on.

   ```
   printf("Hello there");
   ```

3. Print the integer 5 in two different ways.

   Solution:

   ```
   printf("%d\n", 5);
   cout << 5 << endl;
   ```

   ```
   5
   5
   ```

4. Declare and initialize the following variables in as few lines as possible: c as 'a', i as -1000, k as 150, x as 2.5, b as false, and then print them using `cout`.

   Solution:

   ```
   char c = 'a';
   int i = -1000, k = 150, b = 0;
   float x = 2.5;
   // bool b = false;
   cout << c << " " << i << " " << k << " " << x << " " << b << endl;
   ```

```
a -1000 150 2.5 0
```

5. Can you change the program to represent the following statement: "When the sun shines, it's sunny; otherwise, when it rains, it's rainy; otherwise, it's neither sunny nor rainy." Run the program with these values:

   (a) no sun, no rain

   (b) no sun, but rain

   (c) both sun and rain

   Solution:

```
bool sun = false, rain = true;
if (sun == true) printf("It's sunny.");
 else if (rain == true) printf("It's rainy");
 else printf("It's neither sunny nor rainy.");
```

```
It's rainy
```

6. Put this into code using Boolean variables and condition statements: "Socrates is a man. All men are mortal. Therefore, Socrates is mortal."

   Solution:

```
// premise
bool SocratesIsAman = true, menAreMortal = true;
// conclusion
if (SocratesIsAman == true and menAreMortal == true) {
  printf("Socrates is mortal.");
 }
```

```
Socrates is mortal.
```

7. Write a step function that accepts all arguments, not just integer ones. Test it for the sample values -0.5, 0, 0.5. Do not declare and define an extra variable as result in the function, reduce the length of the previously defined function from 7 to 3 lines, and the length of the `main` function from 4 to 2 lines.

   Solution:

```
// define step function
int step(double x) {
  if (x > 0) return 1;
  else if (x == 0) return 0;
  else return -1;
}
// main function
int main() {
  cout << step(-0.5) << "\n"
  << step(0) << "\n"
  << step(0.5) << endl;
  return 0;
}
```

```
-1
0
1
```

8. Print the following constants[4] (and the corresponding headlines) using first `printf` and then `cout`.

```
Print with ‘cout‘:
3.141592654
2.718281828
0.007297352564

Print with ‘printf‘
3.14159265
2.7182818
7.297352564300e-03
```

Solution:

```
// define constants
const double pi = 3.141592653589793; // pi
const double e  = 2.718281828459045; // Euler number
```

[4]These constants are: Pi, the Euler number (exp(1)), and the fine-structure (or Sommerfeld) constant, the strength of the electromagnetic interaction between elementary charged particles.

```
const double a  = 7.2973525643E-03; // fine structure constant
//const double a  = 0.0072973525643; // fine structure constant
// print values with cout
cout.precision(10);
cout << "Print with ‘cout‘:" << endl;
cout << pi << endl
<< e  << endl
<< a  << endl;
cout << endl;
// print values with printf
printf("Print with ‘printf‘\n");
printf("%.8f\n",pi);
printf("%.7f\n",e);
printf("%.12e\n",a);
```

```
Print with ‘cout‘:
3.141592654
2.718281828
0.007297352564

Print with ‘printf‘
3.14159265
2.7182818
7.297352564300e-03
```

9. Create a commented version of the `step` function program written earlier: comment every line of the program, and include a header with multiline comments.

   Solution

```
/* ------------------------------------------ */
/* Define a step function with the values     */
/* f(x == 0) = 0, f(x < 0) = -1, f(x > 0) = 1  */
/* Sample input: x = {-0.5, 0, 0.5}           */
/* Author: Marcus Birkenkrahe (2024)          */
/* ------------------------------------------ */
// define step function:
int step(double x) { // return one integer, take one double argument
  if (x > 0) return 1; // when x is positive, return the value 1
  else if (x == 0) return 0; // when x is zero, return the value 0
  else return -1; // when x is negative, return the value -1
} // end of function
// main function
int main() { // return one integer, take no arguments
  cout << step(-0.5) << "\n" // call step on value, print value, newline
  << step(0) << "\n" // call step on value, print value, newline
  << step(0.5) << endl; // call step on value, print value, newline
  return 0; // return 0 if program ran successfully
} // end of main function
```

```
-1
0
1
```

10. Write a short program for debugging purposes:

    - Remove all header arguments except `C++ :results output`.
    - Before the `main` function: define function `add` that adds two `int`.
    - In `main`, define two sample `int` values 2, 3, then call `add` and print the result using `cout`.
    - Tangle the program as `main.cpp` and compile an executable `add`.
    - Toggle global line number mode
    - Run the debugger on the program.
    - Set breakpoint before calling the function and before printing.
    - Check the values of your variables.
    - Run the program.
    - Step into the function and out of it.
    - Continue and check the values at the next break point.
    - Exit the debugger.
    - Solution code:

      ```
      #include <iostream>
      using namespace std;
      int add(int a, int b) {
      return a+b;
      ```

```
}
int main() {
int x = 2, y = 3;
cout << add(x,y) << endl;
return 0;
}
```

5

- Debugging session script:

```
aletheia@pop-os:~/GitHub/alg1/org$ g++ main.cpp -o add -g
aletheia@pop-os:~/GitHub/alg1/org$ gdb add
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from add...
(gdb) break 6
Breakpoint 1 at 0x11cd: file main.cpp, line 7.
(gdb) run
Starting program: /home/aletheia/GitHub/alg1/org/add
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at main.cpp:7
7         int x = 2, y = 3;
(gdb) info locals
x = -137238220
```

```
y = 32767
(gdb) next
8         cout << add(x,y) << endl;
(gdb) step
add (a=2, b=3) at main.cpp:4
4         return a+b;
(gdb) step
5       }
(gdb) continue
Continuing.
5
[Inferior 1 (process 185899) exited normally]
(gdb) info locals
No frame selected.
(gdb) run
Starting program: /home/aletheia/GitHub/alg1/org/add
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at main.cpp:7
7         int x = 2, y = 3;
(gdb) next
8         cout << add(x,y) << endl;
(gdb) info locals
x = 2
y = 3
(gdb) next
5
9         return 0;
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/aletheia/GitHub/alg1/org/add
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at main.cpp:7
7         int x = 2, y = 3;
(gdb) step
8         cout << add(x,y) << endl;
```

```
(gdb) info locals
x = 2
y = 3
(gdb) step
add (a=2, b=3) at main.cpp:4
4         return a+b;
(gdb) info locals
No locals.
(gdb) step
5       }
(gdb) info locals
No locals.
(gdb) finish
Run till exit from #0  add (a=2, b=3) at main.cpp:5
0x00005555555551ea in main () at main.cpp:8
8         cout << add(x,y) << endl;
Value returned is $1 = 5
(gdb) quit
A debugging session is active.

        Inferior 1 [process 185979] will be killed.

Quit anyway? (y or n) n
Not confirmed.
(gdb) continue
Continuing.
5
[Inferior 1 (process 185979) exited normally]
(gdb) quit
aletheia@pop-os:~/GitHub/alg1/org$
```

Review questions

1. What does Emacs + Org-mode let us do when programming in C++?

   It enables literate programming: we can put documentation and code in one and the same .org file, run code and see the output right away without having to open the command line, and we can export the documentation into multiple formats.

2. Explain the following Org-mode header arguments:

   ```
   :main no
   :namespaces std
   C++
   :includes <cstdio>
   :results silent
   :exports both
   ```

   - Solution:

`:main no`	do not generate 'int main() ...'
`:namespaces std`	use 'cout', 'cin' and 'endl' without 'std::'
`C++`	use C++
`:includes <cstdio>`	#include <cstdio>
`:results none`	do not show any output
`:exports both`	export code and output (e.g. to HTML)
`#+end_src`	end of code block

3. What's the difference between these three statements, and what will the output be?

   ```
   cout << "One" << endl;
   cout << "Two" << "\n";
   std::cout << "Three" << std::endl;
   ```

   ```
   cout << "One" << endl;
   cout << "Two" << "\n";
   std::cout << "Three" << std::endl;
   ```

   ```
   One
   Two
   Three
   ```

   - All three statements print a string and add a new line after it.
   - The first one uses the `endl` keyword from `<iostream>`
   - The second one uses the escape character `\n`

- The third one specifies that `cout` and `endl` are in the `std` namespace.

4. What is a named object also called?

   A variable.

5. What are the logical parts of the statement "The grade will be good when the student studies"?

   - The premise: "The student studies."
   - The conclusion: "The grade will be good."

6. What's the meaning of `public` and `private` states?

   Public states are visible, private states are hidden.

7. What's a "class diagram"?

   A static structure diagram in the Unified Modeling Language (UML) that describes the structure of a system by showing attributes (or states), methods (or behavior), and the relationships among objects, and indicates if attributes and methods are public or private.

8. How do you know the `return` value of a function?

   The first term of the function definition, unless `void`, specifies the `return` value, e.g. `int` for an integer `return` value.

9. How many values can a function `return`?

   A function can only `return` a single value directly, but that single value can be a container that holds multiple values (see example).

   Example:

```
#include <vector>
// function definition
vector<int> count() {
  static int a[5]{100,200,300,400,500}; // declare array of integers
  return vector<int>(a, a + 5); // create and return vector from array
}
```

```
//
int main() {
  // call function and store returned vector in result
  vector<int> result = count();
  // print invidual vector elements using a range-based for loop
  for (int value : result) {
    cout << value << " ";
  }
  return 0;
}
```

```
100 200 300 400 500
```

10. What is pseudocode good for?

    - Understanding your problem before coding a solution
    - Check the logic of your algorithm (in abstracto = without detail)
    - Plan your program (without being bothered by syntax issues)
    - Create text that you can reuse to comment your code

11. What do you have to do to be able to debug a C++ program with `gdb`?

    You have to set the `-g` flag when compiling the program to gather information for the debugger.

Glossary

Term	Definition
main	Program entry point in C++
#include	Preprocessor directive to include libraries
printf	Function to print formatted output
cout	Standard output stream
endl	Add new line
int	Integer data type in
void	Specifies that a function takes no arguments or returns no value
std::endl	Manipulator to insert a newline character and flush the stream
gcc	GNU Compiler Collection for compiling C/C++ programs
Emacs	Editor for creating source code
bash	Shell for running executable files
linker	Tool to link libraries to the source code
compiler	Tool to turn source code into executable code
library	Collection of pre-compiled routines used in programming
Org-mode	An Emacs mode for keeping notes, planning, authoring documents
code chunk	A block of code within a document (Org-mode)
header	The top of a code chunk specifying its parameters (Org-mode)
return	Statement to exit a function and optionally pass back a value
function	A block of code designed to perform a specific task
C++	An object-oriented (OO) programming language.
OOP language	A programming language that supports objects
object	Abstract entity with a state (attribute) and behaviour (method)
state	The attribute or feature of an object, representing what it is
behavior	The method or function of an object, representing what it can do
type	The collection of behaviors and states that describe an object
strongly typed	A language in which each object has a predefined data type
int	Represents the built-in integer type in C++
variable	A named object used to store data
class diagram	Static structure diagram in the Unified Modeling Language (UML)
gdb	GNU debugging program (compile with -g flag)
(gdb) break ln	Set breakpoint at line number
(gdb) run	Run program
(gdb) step	Step into function
(gdb) next	Next statement (do not step into function)
(gdb) continue	Continue execution
(gdb) finish	Finish running the program
(gdb) quit	Exit debugger

Summary

- Requirements for creating and running a C++ program: editor (e.g., Emacs), compiler (e.g., GCC), linker, and shell or IDE with a shell (e.g., bash).
- Every C++ program must have a `main` function and may include preprocessor directives (e.g., `#include`).
- Functions take input (arguments/parameters), can return results, and must be declared before `main`.
- Namespaces are used to prevent naming conflicts. Use the `using` directive to simplify code.
- C++ is an object-oriented language with a focus on state and behavior.
- The `if`, `else`, and `else if` statements are used to make logical choices.
- You can use `cout` (easier) or `printf` (more flexible) for output formatting.
- Comments and pseudocode are important for learning and professional coding.
- You can debug programs with `gdb`, set breakpoints, read out variable values, step in and out of functions. Must compile program with `-g` flag.

References

- This section losely follows chapter 1 (pp. 50-76) of the "C++ Crash Course: A Fast-Pace Introduction" by J Lospinoso (NoStarch, 2019). The section "Comments (and Pseudocode)" uses an example by Smith (2023). "The Rook's Guide to C++" by Jensen (2013) was also used.
- Bastani, Hamsa and Bastani, Osbert and Sungu, Alp and Ge, Haosen and Kabakcı, Özge and Mariman, Rei, Generative AI Can Harm Learning (July 15, 2024). Available at ssrn.com.
- Jensen (2013). The Rook's Guide to C++. URL: rooksguide.org.
- Smith (February 23, 2023). Learn to Write Pseudocode: What It Is and Why You Need It. URL: wikihow.com/Write-Pseudocode.