COMP 1633: Intro to CS II

C++ Basics Continued

Charlotte Curtis

January 15, 2024

Where we left off

- Variable declaration and assignment
- Primitive data types
- Some new C++ operators
- Mixed type arithmetic

Predict the data type

```
5 + i * 2
d + i * 2
d / 9.33
7 / i
7.0 / i
42 + 7 / (i * 1.2)
```

Today's topics

- Named Constants
- Comments
- Input/Output
- Type casting
- Debugging with gdb

> *Textbook Sections 2.2, 2.5*

A few new operators

Like Python, C++ has the compound assignment operators `+=`, `-=`, `*=`, `/=`, and `%=`. There's a few new ones as well:

- `++` and `--`: increment and decrement by 1
  - Can be either `++x` or `x++`
- **Unary** operators: `+` and `-`
  - Finally you can write `x + -5` instead of `x - 5`!
- `++` and `--` happen first, then unary operators, then the usual BEDMAS

A full listing of operator precedence can be found here

Constants using `const`

In Python, constants are just a **convention**:

```python
PI = 3.14159
GST = 0.05
NUM_PLANETS = 8
```

In C++, use the keyword `const`:

```cpp
const double PI = 3.14159;
const double GST = 0.05;
const int NUM_PLANETS = 8;
```

- `const` is a **modifier** that prevents the value from being changed

> *C++ has a number of modifiers that make the compiler enforce rules, turning run-time or logic errors into compile time errors (a good thing!)*

Comments

- C++ has two types of comments:
  - Single line comments: `//` (most common)
  - Multi-line comments: `/* */`
  - Be consistent!
- Stylistically, comments should be used the same way as Python
  - Explain **why** you are doing something, not **what** you are doing
  - Use self-documenting variable names and code structure
  - Short comments in line with code are okay, but stick to a max of ~80 characters per line total

Displaying output

Assuming `#include <iostream>` and `using namespace std;` we can display output with:

```
cout << "Hello World!\n";
```

- `cout` is the **standard output stream**
  - A **stream** is a source or destination of characters of indefinite length
- `<<` is the **stream insertion operator**
- We're telling C++ to "insert `"Hello World!\n"` into the output stream

> *Unlike Python, we need to explicitly add `\n` or `endl` to get a new line*

More output magic

Like Python's `print`, C++ is happy to mix and match types:

```
int age = NOT_TELLING;
cout << "I am " << age << " years old.\n";
```

You can insert as many things in the string as you like, and even break over lines:

```
cout << "This is a very long string that I want to break over "
     << "multiple lines.\n"
     << "This is on the next line.\n";
```

- String literals **cannot** be broken over lines
- Only one statement means only one semicolon

Reading input

Like the standard output `cout`, C++ has a standard input `cin`:

```
cin >> variable_name;
```

- `>>` is the **stream extraction operator**
- `cin` will wait for the user to type something and press enter
- `variable_name` must be declared, and must match the **data type** of the input

```
int age;
cout << "Enter your age in years: ";
cin >> age;
```

The `cin` input stream

Like `cout`, `cin` can be used to read multiple values:

```
char first_initial, last_initial;
int year, age;
cout << "Enter your first and last initials: ";
cin >> first_initial >> last_initial;

cout << "Enter your program year and current age: ";
cin >> year >> age;

cout << "Thanks, " << first_initial << last_initial
     << "! You were " << (age - year) << " when you started!\n";
```

Check-in 1/2

True or false:

Like Python, C++ will include a prompt for the user when requesting input.

A. True

B. False

Check-in 2/2

True or false:

Multiple inputs can be separated by whitespace.

A. True

B. False

Buffered input

- Typed input is read and stored in a **buffer** (temporary storage)
- This allows the user to backspace and make corrections before submitting
- `cin` follows (approximately) this process:

```
if the buffer is empty
    read from the keyboard
else
    process next value in the buffer
```

- The **data type** of the variable to the right of `>>` determines how the input is interpreted

Type-dependent input processing

Data Type	Input Processing
`int`	Read all characters until a non-digit is found
`double`	Read all characters until a non-digit or non-decimal is found*
`char`	Read the next character

- For all data types, leading whitespace is ignored and multiple whitespace characters are treated as a single delimiter
- **Important**: the last character (often `\n`) is left in the buffer

* Or scientific notation, e.g. 2.99e8

Type casting

Mixed type arithmetic can result in **implicit type casting**:

```
int i = 1;
double d = (1 + i) * 3.4; // ok
d = i; // still okay
i = d; // compiler warning!
```

- Best to be **explicit** with `static_cast`:

  ```
  i = static_cast<int>(d);
  ```

- General syntax: `static_cast<type>(expression)`

Type casting check-in

In the following code sample, what is the final value of `pi_i`?

A. 0

B. 1

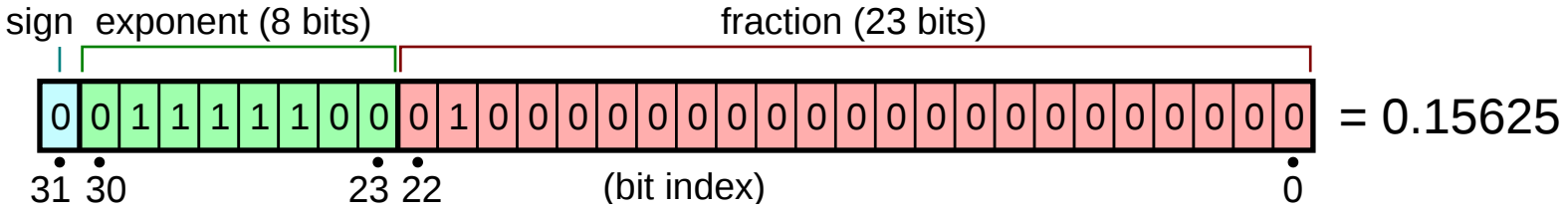
C. 2

D. 3

```
double pi = 3.14159;
int pi_i = static_cast<int>(pi / 2);
```

Limitations of `double`

- Declaring a `double` allocates 8 bytes or 64 bits of memory (32 bit `float` shown below):



- Allows for numbers up to 1.8×10^{308}, but "only" 15-17 digits of precision
- Doubles are **inexact**: `3 * 0.1 == 0.3` may evaluate to `false`!

Image from https://en.wikipedia.org/wiki/IEEE_754

Limitations of `int`

- Declaring an `int` allocates 4 bytes or 32 bits of memory
- This allows for storing numbers up to $2^{31} - 1$ or $2,147,483,647$

> *Why not 2^{32}?*

- Integers are **exact**, so can be safely used for equality comparisons
- BUT if you exceed the maximum value, you get **integer overflow**:

```
int i = 2147483647;
i = i + 1;
cout << i << endl; // -2147483648
```

Debugging with gdb

- In tomorrow's lab, you will be introduced to the GNU Debugger `gdb`
- `gdb` is a command-line tool that allows you to:
  - Run your program line-by-line
  - Inspect the values of variables
  - Set breakpoints to pause execution
  - And much more!
- To build with **debug info** (such as line numbers) use the `-g` flag:

```
g++ -g hello.cpp
```

gdb demo

- After building with `-g`, run `gdb` on the executable:

```
gdb ./a.out
```

- You will see a `(gdb)` prompt
- Type `run` to start the program - this will run the whole thing
- Type `list` to see the source code
- To add a breakpoint, type `b <line number>` (or `break <line number>`), e.g.:

```
b 7
```

- Now run again, and the program will pause at line 7

Basic gdb commands

Command	Description
`run`	Run the program
`list`	List the source code
`b <line number>`	Set a breakpoint at the given line number
`d <breakpoint number>`	Delete the given breakpoint
`n`	Execute the next line of code
`p <variable name>`	Print the value of the given variable
`c`	Continue execution until the next breakpoint

Coming up next

- Lab: C++ and gdb
- Lecture: Using and defining functions in C++

> *Textbook Chapter 4 and start of 5*