Variables and Datatypes

Ajit Rajwade

Refer: Chapter 3 of the book by Abhiram Ranade

Variables

- A **region of memory** for holding some piece of data is called a **variable**.
- C++ allows you to create as many variables as you want (within limits of the memory) and **name** them as you wish.
- The name of a variable is also called **identifier**.
- Each variable will be referred to by this identifier right through the program.
- How do you declare a variable? By writing a statement such as:

```
<data-type> <variable-identifier>;
```

- Example declarations seen earlier: `int num_sides; int side_length;`
- Each variable has a certain **data-type**.
- Each variable is stored in a memory location called its **address**.

Data-type and memory sizes

- Each data-type occupies a certain amount of memory space - the **size** of the data-type.
- The memory of a typical computer contains basic units - capacitors in a state that is on or off.
- Each such unit stores value called a **bit** which either 0 (off) or 1 (on).
- Memory is often measured in terms of **bytes**, i.e. a collection of 8 bits.
- 1 **kilobyte** (KB) = 1024 bytes = 2^10 bytes
- 1 **megabyte** (MB) = 1024 KB = 2^20 bytes
- 1 **gigabyte** (GB) = 1024 MB = 2^30 bytes

Variable Names (Identifiers)

- Contain upper and lower case alphabets, digits and the underscore character ‘_’
- They **cannot** begin with a digit, but can begin with the underscore.
- Variable names are case sensitive, i.e. ‘A’ and ‘a’ are **different** identifiers.
- Some words such as `int` or `include` are reserved keywords by C++ - they cannot be used as variable names.
- You have the freedom to name your variables - but use meaningful, simple and descriptive names.
- Examples of **legal** identifiers: a, abc, A, ABC, A12; A_B123, A23B56, etc
- Examples of **illegal** identifiers: int, #a, 1abc, @abc, a@, etc.

Data-type	Range of values	#bytes allocated	Purpose
`unsigned char` (`char` by default is `unsigned char`)	0 to 255	1	characters , small-valued non-negative integers
`signed char`	-128 to 127	1	characters , small-valued integers

Data-type	Range of values	#bytes allocated	Purpose
`short int`	-2^{15} to $+2^{15}-1$	2	Medium-valued integers
`unsigned short int`	0 to $2^{16}-1$	2	Medium-valued non-negative integers
`int` (by default `int` is `signed int`)	-2^{31} to $2^{31}-1$	4	Standard integers
`unsigned int`	0 to $2^{32}-1$	4	Standard non-negative integers
`long int`	-2^{31} to $2^{31}-1$	4	Standard integers
`unsigned long int`	0 to $2^{32}-1$	4	Standard non-negative integers
`long long int`	-2^{63} to $2^{63}-1$	8	Large-valued integers
`unsigned long long int`	0 to $2^{64}-1$	8	Large-valued non-negative integers
`bool`	0 (false) or 1 (true)	1	Logical values

int, long int and long long int

- In some compilers/OS combinations, long int is allocated 8 bytes, the same as long long int.
- In others (including the one installed in the lab machines), int and long int are allocated 4 bytes whereas long long int is allocated 8 bytes.
- In general, do not get bogged down by these details.
- The memory requirement for any data-type can be obtained using `sizeof (<datatype>)`. For example: `cout << sizeof(long long int);`

Representing integers in a computer

- We are used to numbers in the decimal system, which use 10 as the base.
- Thus a number like 6237 = $6 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$.
- But a computer stores numbers in base 2 (binary representation).
- 6237 = 1100001011101 in binary format
- 6237 = $1 \times 2^{12} + 1 \times 2^{11} + 1 \times 2^6 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0$
- To obtain the binary representation of a number, divide it by 2 and note the quotient and remainder.
- Keep dividing the quotient by 2 until the quotient becomes 0. Each time keep the remainder in mind.
- Then just list out the remainders in reverse order.

Representing integers in a computer

- Example: 12
- 12/2 = 6 remainder 0
- 6/2 = 3 remainder 0
- 3/2 = 1 remainder 1
- 1 / 2 = 0 remainder 1
- The binary representation is 1100
- Example: 19
- 19/2 = 9 remainder 1
- 9/2 = 4 remainder 1
- 4/2 = 2 remainder 0
- 2/2 = 1 remainder 0
- 1 / 2 = 0 remainder 1
- The binary representation is 10011

Representing integers in a computer

- An `int` data-type (same as `signed int`) has 32 bits.
- For the `int` data-type, one bit is reserved for the **sign**.
- The remaining bits (31 of them) are used for storing the number.
- Hence the range of value a `signed int` can acquire is as given in the table earlier.
- For an `unsigned int`, only non-negative numbers can be stored.
- Hence there is no need for a sign bit.
- Thus all 32 bits are used for representing the number and its range is as per what is given in the table earlier.

Data-type	Range of values	#bytes allocated	Purpose
`float`	7 digits of precision ~+/- 10^{-38} to 10^{38}	4	Floating point (real-valued, decimal) numbers: positive or negative
`double`	15 digits of precision ~+/- 10^{-308} to 10^{308}	8	Floating point (real-valued, decimal) numbers: positive or negative with even higher precision
`long double`	18 digits of precision +/- 10^{-4932} to 10^{4932}	12	Floating point (real-valued, decimal) numbers: positive or negative with even higher precision

All three data-types above are called floating point data-types.
They are used to store decimal fractions and are heavily used in numerical computations and graphics.
They are stored in the computer in a peculiar manner which we will not cover in this course.

Sample declarations

```
char firstLetterName;

int telephone_number; int telephoneNumber;

unsigned int number_leopards_SGNP;

float  area_square;

char first_letter, last_letter;

int telephone_number, house_number;

float area_square, volume_cube;
```

Multiple variables of the same datatype can be declared in one statement

Variable Initialization

- Upon creation, you can force a variable to have a certain value.
- Examples:

```
int p = 10;

float pi = 3.142857;

char firstLetter = 'b';
```

- All characters have associated ASCII values (from 0 to 255) - which you can find at https://en.wikipedia.org/wiki/ASCII .
- Lower case alphabets have ASCII values from 97 ('a') to 122 ('z').
- Upper case alphabets have ASCII values from 65 ('A') to 90 ('Z').

`const` variables

- These are variables whose value is not intended to ever change throughout a program.
- Examples:

```
const float pi = 3.142857, avogadro = 6.022E23;
```

- If your program attempts to change the value of a `const` variable, the program won't compile - syntax error.

`cin` and `cout` again

- `cin` is used to read values entered through a keyboard into a variable
- Can be used for one or more variables
- Examples:

```
cin >> num_sides;
```

```
cin >> num_sides >> side_length;
```

- Printing variable values is performed by `cout`.
- Examples: (`endl` is used to print onto a new line)

```
cout << num_sides << side_length << endl;
```

Assignment statements

- A variable can be assigned a value via the = operator, also called an **assignment operator**
- The assignment to the variable can be a single value or an expression.
- Examples:

```
int p=3,q;  // p gets a value of 3; q can have an arbitrary value

q = p*p+1;
```

- The general syntax is `variable = expression;`
- The expression on the right gets evaluated and then assigned to the variable on the left.
- There can be compound assignments: `int x,y,z; x = y = z = 1;`
- Here `x`, `y`, `z` will all acquire the value 1. The rightmost assignment is evaluated first.
- Note that = is the assignment operator. It is not the equality sign. For example, `p = p+1;` is valid in C++ as it increases the value of `p` by 1, but mathematically false.

Expressions and Operators

- An expression consists of variable and operators
- Operators: + (addition), − (subtraction), * (multiplication), / (division), % (modulo, i.e. remainder upon division)
- +,−,*,/ work with floats and integer operands, whereas % works with only integer operands
- Examples of expressions:

```
x+y

x%4

c*c/d + e + 5
```

Subtleties of /

- Be careful of the / operator used for division!
- Example:

```
int a = 10, b = 20, c, d; float e;

c = b/a; // will acquire the value 2

d = a/b; // will acquire the value 0 and not 0.5

e = a/b; // will acquire the value 0 and not 0.5
```

- The / operator gives only the integer part of the quotient if the divisor and dividend are both integers!
- How to avoid this? **'Cast'** either divisor *or* dividend or both to floats:

```
e = (float)a/(float)b; // will acquire the value 0.5
```

Equivalent syntax:

```
e = float(a)/float(b); // will acquire the value 0.5
```

Subtleties of /

- To evaluate C(100,6) = 100 x 99 x 98 x 97 x 96 x 95 / (1 x 2 x 3 x 4 x 5 x 6), which one(s) of the following will work?

```
int x = 100*99*98*97*96*95/(1*2*3*4*5*6);

int y = 100/1 * 99/2 * 98/3 * 97/4 * 96/5 * 95/6;

int z = 100.0*99*98*97*96*95/(1*2*3*4*5*6);

int w = 100.0/1 * 99/2 * 98/3 * 97/4 * 96/5 * 95/6;
```

The concept of overflow

- No digital computer can store integers of infinite value - there is always a fixed range.
- For example, `int` stores values between -2^{31} to $2^{31}-1$ (=2147483647).
- Now consider the following program:

```
int a = 2147483647;

a = a+1;

cout << a; // this will print -2147483648 !!
```

- Why does this happen? Because `int` uses only 32 bits, out of which 1 bit is for the sign (+1 or -1) and the remaining 31 are for the value.
- Since 2147483647 = $2^{31}-1$, further addition by 1 does not increase its value but wraps it down to the lowermost number `int` can store.
- This is called **numerical overflow** or **wrap-around error**.
- What will happen in case `a` above was an `unsigned int`?

The concept of roundoff errors

- There is a limit to how many digits after the decimal point a floating point data-type can store.
- Example:

```
float avogadro = 6.022E23; // 6.022 x 10^23

float y = 1.5, w;

w = avogadro + y;
```

- The value of `w` will surprisingly be equal to `avogadro` even though we added 1.5.
- Why? Because a float can store only 7 decimal digits. Anything beyond that is **rounded off to zero**.
- This is called **roundoff error**.

Programming: compute the average of given numbers

- Aim to take some `n` numbers from a user
- And compute their average and print it on the screen
- In fact, we will take even `n` (the number of numbers) as input from the user

```
main_program{
int n,a;
float avg = 0.0; // initialize average to 0 – very important!
cout << "how many numbers?"; cin >> n;
repeat(n){
    cout << "enter the next number"; cin >> a;
    avg = avg + a;
}
avg = avg/n;  // at the end of the loop, avg contains the sum; divide //
by n to get average
cout << avg;
}
```

Can you modify this to compute the maximum of the given numbers instead of the average? Use statements such as `maxval = max(a,b);` where `a` and `b` are two suitable numbers and `max(a,b)` simply returns their maximum.

Concept of accumulation

- Notice how `avg` was initialized to zero in the previous program.
- After each number is entered, `avg` is incremented by the value of that number.
- This continues for `n` numbers.
- At the end of the inner loop, `avg` reflects the sum of those numbers and is finally divided by `n` to give the average.
- This process is called **accumulation**, as the sum of the numbers gets **accumulated** each time.
- Can you use this concept to compute the **maximum** of the `n` numbers? (see previous slide). How about the **minimum**?

Concept of accumulation: factorial computation

- We know that factorial(n) = n x (n-1) x (n-2) x ... x 2 x 1.
- Let us write a program in C++ to compute the factorial of a number `n` entered by the user.
- In the earlier case, the sum was initialized to 0 and we kept adding numbers to this sum (to accumulate).
- Now, we need to maintain a **product** which is initialized to 1, and we keep multiplying numbers to the product.
- We are still using the accumulation method - and also something else called **sequence generation**.

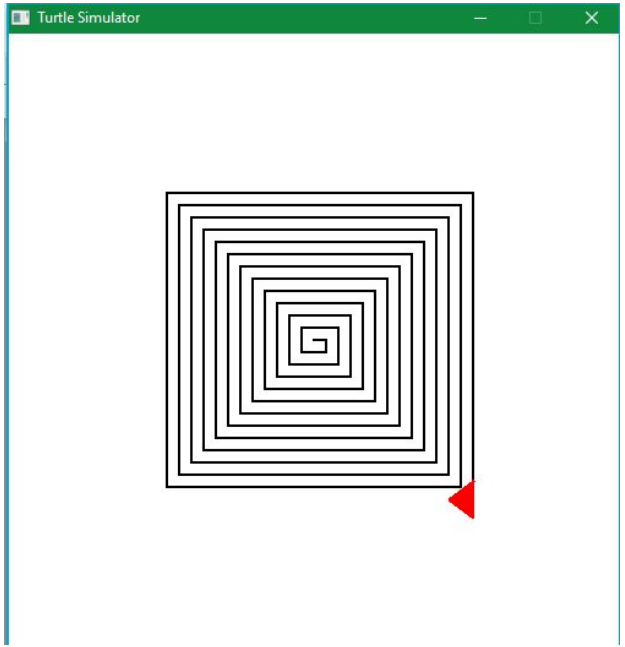
Concept of accumulation and sequence generation

```
main_program{
int n,i=1; // i is a useful counter – we will see soon
int facval = 1; // initialize factorial value to 1 – very important!
cout << "Enter the number whose factorial you want to compute: "; cin >> n;
repeat(n){
     facval = facval * i; // accumulation
     i = i+1; // sequence generation
}
cout << facval;
}
```

The variable `i` starts of with the value 1 and its value is incremented by 1. This called **sequence generation**.

Drawing a Spiral

- Now we will use the sequence generation principle in drawing a spiral of the following form using turtle graphics.



Drawing a Spiral

- Notice that the spiral is made up of L-shaped patterns, though the L can be rotated left or right by 90 degrees.
- Moreover as you move outwards, the length of both segments of the L pattern are increasing in a specific manner.
- This is sequence generation!
- To print out one segment of the L, the turtle needs to go forward by some `x` pixels.
- Then it needs to turn by 90 degrees and go forward by the same amount to finish the second segment of the L.
- Thereafter, we need to print another L, but this time the length of the segments must increase.
- And we keep printing out more and more L patterns of larger size.

Drawing a Spiral

```
main_program{
turtleSim(); // start the turtle simulator!
int i=1; // a counter
repeat(25){ // this is the number of L's we want
    forward(i*10); right(90); // one segment of the L
    forward(i*10); right(90); // the second segment of L
    i=i+1; // increment the counter
}
wait(20);
}
```

Program Blocks

```
main_program{

int n;

float avg = 0.0;  // initialize average to 0 - very important!

cout << "how many numbers?"; cin >> n;

repeat(n){

    int a;

    cout << "enter the next number"; cin >> a;

    avg = avg + a;

}

avg = avg/n;   // at the end of the loop, avg contains the sum; divide by n to
get average

cout << avg;

}
```

Program Blocks

- The variable `a` here is declared inside the repeat loop.
- It gets created when the repeat loop begins and is available for use only until the end of the `repeat` loop.
- After the `repeat` loop is over, the variable `a` is **destroyed** (i.e. memory earmarked for `a` is now available for use with other variables) and attempting to use it after the `repeat` loop will cause a **compilation error.**
- We say that the **scope** of `a` is the **block** of code involving the `repeat` loop.
- A **block** is any piece of code between `{` and `}`.
- Blocks can be **nested** inside each other.

Program Blocks

```
main_program{
int n;
int a = 10;
float avg = 0.0;  // initialize average to 0 – very important!
cout << "how many numbers?"; cin >> n;
repeat(n){
     int a;
     cout << "enter the next number"; cin >> a;
     cout << a;
     avg = avg + a;
}
cout << a; // this will print the value of 10.
avg = avg/n;   // at the end of the loop, avg contains the sum; divide by n
to get average
cout << avg;
}
```

- Consider there are two variables with the same identifier in a parent block and a child block - in this case `a`.
- Inside the child block, `a` will refer to the variable declared inside the child block - called **shadowing**.
- Once the child block exits, `a` will then refer to the one declared inside the parent block.

Increment and Decrement Operators

- During sequence generation we have seen statements like `i=i+1;`
- There is another statement equivalent to it: `i++;`
- Here `++` is called an **increment** operator. Likewise there is also a decrement operator `--`. Both these act only on **integers** and **characters** (**not** float or bool).
- These operators can be used in expressions and assignments, for example:

```
int z = 1, k; k = z++;
```

- In the above, `k` is assigned the value of `z`, **after** which `z` is incremented by 1. This is hence called a **post-increment operator**.
- Now consider: `int z = 1, k; k = ++z;`
- In the above, `z` is **first** incremented by 1, and then `k` is assigned the new value of `z` (i.e. 2). This is hence called a **pre-increment operator**.
- Analogously, there are **post-decrement** and **pre-decrement** operators.

Compound assignments

- Consider statements like: `x = x*y;` or `x = x*2;`
- These statements can be equivalently be written as: `x *= y;` or `x*= 2;`
- Likewise the statements: `x = x+y;` or `x = x+2;` can be equivalently be written as `x+= y;` or `x+= 2;`
- The operators +=, *=, and likewise −=, /= are called **compound assignment** operators.

Program debugging

- Suppose you write a program which compiles properly, but does **not** produce the desired output when you execute it.
- Obviously, you made some **error**!
- How does one **trace** the error?
- You can **dry-run** the program, i.e. execute it step by step with pen and paper.
- This is tedious and **not recommended** for large programs.
- You can **print** the values of variables at various points.
- But this can really create a **cluttered** display and is not recommended for large programs. Moreover, for every print statement you add, you need to re-compile.
- So, you use a **debugger**.
- A debugger allows you to run the program line by line, and watch the values of all variables.
- It allows you to **see** what is going on in your program, besides giving you insights to **correct** program errors.
- In fact, it does much more than just that!

Program debugging

- The well-known debugger on Linux is called **gdb** (GNU debugger).
- In order to debug a program, you must compile it in **debug mode**.
- That is, instead of running the command `g++ prog1.cpp -o a.out` in the Linux shell, you should run the modified command `g++ prog1.cpp -g -o a.out`
- The flag `-g` tells the g++ compiler that it must compile in debug mode.
- Then, in order to run the program in debug mode, you execute the command `gdb a.out` in the Linux shell.
- `gdb` has its own shell which looks like what you see on the next slide.

```
ajitvr@surya:~/Programs_CS101$ gdb a.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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:
<http://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 a.out...done.
(gdb) break 6
Breakpoint 1 at 0x4008cd: file factorial.cpp, line 6.
(gdb) run
Starting program: /users/fac/ajitvr/Programs_CS101/a.out

Breakpoint 1, main () at factorial.cpp:6
6               int k,n,i=1; // i is a useful counter - we will see soon
(gdb)
```

GDB commands

`run`: just runs the program **from the beginning** via the gdb shell.

`break <line_number>`: ensure that when the program runs, it stops when so and so **line number** is encountered - in this case 6. The line number is called a **breakpoint**. The line number is taken from your .cpp file.

`cont`: runs the program from the **current line** (via the gdb shell) until the next breakpoint

When program execution halts at a breakpoint, you can examine values of variables via `print <variable_name>` command. You can even set variable values via `set <variable_name>`

```
Hardware watchpoint 2: facval
(gdb) watch i
Hardware watchpoint 3: i
(gdb) n
Enter the number whose factorial you want to compute: 7
11              for(k=0;k<n;k++){
(gdb) n
12                      facval = facval * i; // accumulation
(gdb) n
13                      i = i+1; // sequence generation
(gdb) n

Hardware watchpoint 3: i

Old value = 1
New value = 2
main () at factorial.cpp:11
11              for(k=0;k<n;k++){
(gdb) n
12                      facval = facval * i; // accumulation
(gdb) n

Hardware watchpoint 2: facval

Old value = 1
New value = 2
main () at factorial.cpp:13
13                      i = i+1; // sequence generation
(gdb) n

Hardware watchpoint 3: i

Old value = 2
New value = 3
main () at factorial.cpp:11
11              for(k=0;k<n;k++){
(gdb)
```

More GDB commands

`n`: runs the next step (line) of the program

`watch <variable_name>`: halts program execution when the value of `<variable_name>` **changes**, and it prints out the most recent and changed values of `<variable_name>`. This is called a **watchpoint**. It can also be used for an expression instead of a single variable or for a whole set of addresses or memory locations.

Try and use gdb yourself for the factorial program!

Online GDB (quick) guide
https://web.eecs.umich.edu/~sugih/pointers/gdbQS.html

Tips for using `gdb`

- You must compile the program using the -g flag using `g++ prog1.cpp -g -o a.out`
- When you enter the gdb shell, you must put down a breakpoint at some desired line and only then execute `run`, so that the program halts at the chosen line number.
- Otherwise the program will just execute when you enter `run`, inside the gdb shell, without giving any control to you.

Mathematical Functions

- C++ provides you a rich set of mathematical functions.
- All or most of the following use floating point parameters and give out floating point values.

➔ `sqrt(x)`

➔ `cos(x), sin(x), tan(x), sinh(x), cosh(x), tanh(x)`

➔ `log(x)` (base e), `abs(x), fabs(x), exp(x), pow(x,y)`

➔ `ceil(x), floor(x)`

- You can write programs which use these functions!
- These functions exist in the `cmath` library which you include via `#include <cmath>` at the top of your program.
- If you use `#include<simplecpp>`, they are included by default.

Example: Mathematical program

- The following identity in calculus is well known: $\int_1^e \frac{1}{t}\,dt = 1.$
- Here `e` is Napier's base.
- Integration can be approximated by summation, and we will write a computer program to perform the addition.
- We will divide the interval from 1 to `e` into some `N` sub-intervals (x_0, x_1), $(x_1, x_2), \ldots, (x_{N-1}, x_N)$ where $x_N = e$, $x_0 = 1$ and we use the well-known mid-point rule in numerical integration.
- **Midpoint rule:** Let a function `f(x)` be defined on the closed interval [a,b] that is subdivided into `N` sub-intervals of length equal to `(b-a)/N` using `N+1` points $(x_0, x_1, x_2, \ldots, x_{N-1}, x_N)$. Then we have:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} f\left(\frac{x_{i-1}+x_i}{2}\right)\frac{b-a}{n}$$

- In our case `a = 1, b = e, f(x) = 1/x`.

```
main_program{
float a, b, x1, x2, delta, sumval = 0.0; // initialize summation to 0
int N = 1000; // number of sub-intervals
a = 1;
b = exp(1); // value of 'e'
x1 = a; // stands for x_{i-1} in the formula on the previous slide
delta = (b-a)/N; //sub-interval width
x2 = a+delta; // stands for x_i in the formula on the previous slide
repeat(N){
sumval = sumval + delta*2/(x1+x2); // invoking the midpoint rule for f(x) = 1/x
x1=x2; // changing the values of x_{i-1} and x_i
x2 += delta;
} // ends repeat loop
cout << sumval; // you will see this evaluates to 1
}
```

Example: Mathematical program

- The earlier method can be used to compute the logarithm (to base `e`) of any positive `x`.
- Using the following fact, modify the previous program to accomplish this:

$$\ln x = \int_1^x \frac{dt}{t}$$