Chapter 2: C++ Basics

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Variables and Assignments

- Variables are named memory locations to store data in
- Think of them like a box that you can store data in
  - We put a number (data) in the box
  - We can change the number (data) in the box
  - We can remove (erase) the number (data) in the box
- You refer to a particular box through a name (called an *identifier*)
- Variables must be declared
  - Variable declarations tell the computer "I want a box named `my_variable_name` to store data of a particular type"

Basic datatypes

Datatype	Description
`int`, `short`, `long`	whole (integer) numbers
`double`, `float`	decimal numbers
`string`	characters
`bool`	`true` or `false`
`char`	a single character

Identifiers

- Identifiers are the names of variables
- Identifiers must:
  - Begin with `a-z`, `A-Z`, or `_`
  - Followed by `a-z`, `A-Z`, `0-9` or `_`
- Identifiers are case-sensitive
- It is best practice to initialize variable values when they are declared

Declaring variables

- Before you can use a variable, you must declare it. Declaring a variable tells the compiler the type of data to store and allocates memory (a box to store data in) in computer memory.
- Examples:
  - `int counter; //declares one integer named counter`
  - `double weight, height; //declares two doubles, one named weight, another named height`

Assignment statements

- Assignment statements change the value of a variable.
  - Example: `counter = 5; //sets the variable counter to be the value 5`
- We can also assign variables to the value of other variables:
  - Example: `counter = b;`
- The variable that will be changed is always on the left of the assignment operator =
- The righthand side of an assignment can be:
  - A constant (`age = 21;`)
  - A variable (`valueA = valueB;`)
  - Expressions (`valueA = valueB * 10 + valueC;`)
- Note that the assignment operator is not the same as an algebra statement - we are not saying "the left-hand side is equal to the right-hand side." We are saying "the variable on the left-hand side now contains the value of the right-hand side."
- We we *initialize* a variable, we assign it to a value as we declare it
  - Declaring a variable does not give it a value - we must either initialize the variable as we declare it, or assign the variable later

```
int a = 5; // this initializes a to be 5
int b;     // this declares the variable b
b = 7;     // this assigns b the value 7
```

Keywords

- Keywords have special meaning to C++; they are reserved for the language itself

`break`	`case`	`char`
`const`	`default`	`do`

`double`	`else`	`extern`
`float`	`for`	`if`
`int`	`long`	`return`

- We will learn more about these keywords as we go, but you cannot use a keyword for anything other than it's purpose as a keyword (i.e. you cannot use a keyword as an identifier)

Input and Output in C++

- Input and output allows you to get information from the user (input) and display information to the user (output)
  - We have already seen output with our `cout` statements in `hello_world.cpp`
- C++ input statement:
  - `cin >> number;`
  - Value is extracted from the keyboard and stored in the variable called `number`
- C++ output statement:
  - `cout << "Hello, world!";`
  - Sends information from program to the terminal screen
  - Double quotes `"..."` delimit a *string*
  - `\n` sends a *newline character* to the terminal (carriage return)
    * The \ character indicates we are entering an *escape sequence*. Other notable escapes sequences include `\t` (tab), `\\` (backslash), `\"` (qoute character)
    * We must escape these characters because they have a special meaning in C++

Formatting numbers

- When we `cout` floating point numbers (**`double`** or **`float`**), we may wish to specify the precision of the output
- Suppose we wanted to output a dollar amount, like $35.40

```
double price = 35.4;
cout << "The price is $" << price << endl;
```

- The output could be: `The price is $34.4`, `The price is $34.400000`, `The price is $3.4400000e1`.
- If we want to output $34.40, we need to tell the compiler what level of precision to use

```
cout.setf(ios::fixed); // specify fixed point notation
cout.setf(ios::showpoint); // specify the decimal will always be shown
cout.precision(2); // specify that two decimal places will always be
    shown
// cout is now ready to print to two decimal places
cout << "The price is " << price << endl;
```

Prompts

- It's very common to prompt for a value with `cout` and then receive it with `cin`
  - Many homework assignments will expect you to do something similar!

```
cout << "Enter your age: ";
cin << age;
```

Literals

- A literal, fixed, static value used in a program
- Literals are invariant values. They can never be changed. They can never store data.
- Literals are literally some value.
- `2` is the literal integer value 2. `2.0` is the literal floating point value 2.0
  - Note that these are actually different types of literals

Literals vs. Variables

- A literal is a fixed values that never changes
- A variable is a container for data
  - A variable can change its data over time
  - A variable can only hold one value at a time
- A literal has no "box" (memory) to store data in; a literal is a fixed value embedded in your program.
- This means that a literal can **never** be on the left-hand side of an assignment statement
  - `5 = 6;` is not possible because `5` is a literal. It is not a box (variable) that we can store data in
  - Always think about the left-hand side of the assignment statement in terms of boxes that you can store data in
  - Literals are literal values and not boxes to store data in, and therefore cannot be on the left-hand side of an assignment statement.

Basic datatypes (revisited)

Datatype	Description
`int`, `short`, `long`	whole (integer) numbers
`double`, `float`	decimal numbers
`string`	characters
`bool`	`true` or `false`
`char`	a single character

`int`

- Stores an integer value.
- Typically stored in 32 bits (the computer uses 32 bits to represent the number)
  - If you have a set of integers centered around 0, what's the maximum and minimum integer you can represent with 32-bits?
    * 32 bits leads to 4294967296 which is 2^{32} (binary is base 2, and we have 32 bits)
    * Maximum value: `+2147483647`
    * Minimum value: `-2147483648`
- Example usage:

```
int a = 5;
```

`char`

- Capable of holding any member of the character set.
- Stored in 1 byte (8 bits).
- The underlying structure has the same type of data as an `int` (with a smaller range of data)
  - However, we the way we *should* use chars is not through integer references
  - This is all because internally a character is literally an integer to the computer
- Examples of characters:

```
'a'
'b'
'3'
'\0'   // null character
'\n'   // newline character
```

```
'\t'   // tab character
```

`string`

- A **string literal** is a collection of characters in a single string
  - `"Hello, world!"` is an example of a string literal
  - String literals are denoted by " instead of ' for their wrapping quotations
- In C++, a string is class, which is different from the other primitive datatypes we've discussed thus far.
- Strings are used to store collections of characters.
- You can use strings by including the string library with `#include <string>`
  - Then you can declare and initialize a string with `string name = "Mark Edmonds";`

`float`

- Holds a floating point number, such as `32.2`
- All representations of floating point numbers are inexact.
- Adding `f` to the end of a number indicates it is to be interpreted as a float
- Examples of floats:

```
32.3
3223.64563f
4.0f
6.022e+23f
```

`double`

- Exact same as a **`float`**, but uses double the precision (i.e. double the computer memory) to store the data

Type Modifiers

- We may want to modify the amount of storage used by a type.
- This enables data to use more or less memory depending upon the use case.
- Adding a modifier of **`long`** will make the type use more memory
- Adding a modifier of **`short`** will make the type use less memory
- Adding a modifier of `unsigned` will make the type non-negative in all cases (changes the range of possible values

- If you use **short** or **long** by itself, the **int** type is implied

```
unsigned short int usi;   /* fully qualified -- unsigned short int */
short si;                 /* short int */
unsigned long uli;        /* unsigned long int */
```

- The **const** makes a particular variable constant, or unmodifiable.
  - You *must* initialize the value when you declare it.
  - What's the advantage?
    * You gain additional protections against a programmer making a mistake and modifying a value they shouldn't
    * Also protects against magic numbers - don't put the same literal all over your program. Use a constant to define the value once and use the constant everywhere you need that value

Type Compatibility

- Intuitively, it makes sense that we'd be able to convert between certain types.
  - For instance, converting between an integer and a floating point number seems like a reasonable thing to do
  - There are a few rules regarding these implicit type conversions, so we should avoid using them unless it's really justified
- Implicit type conversions:
  - **int** <-> **double** conversion: when converting to an **int**, the decimal place will **not** be rounded; it will be *truncated*
    * This means writing something like `int a = 2.4;` will result in `a` holding the integer 2. Similarly, writing something like `int b = 2.7;` will also result in `a` holding the integer 2.
  - **int** <-> **float** conversion: same as **int** <-> **double**.
  - **char** <-> **int** conversions: characters are really stored as integers, but there's no semantically meaningful way to convert between them.
    * For instance, we could write `int value = 'A';`, but this integer has no semantic meaning
    * It's best to avoid this at all costs - there's really no reason to convert between an **int** and a **char**
  - `bool` <-> **int** conversion: any non-zero integer is evaluted to **true** and the integer`0` is evaluated to **false**
    * If we convert **true** to an integer, we get the integer `1`
    * It's also best to avoid this - there's no reason to convert between an **int** and a `bool`

Basic Arithmetic Operators

- C++ supports basic arithmetic operators to help you do math.
- Basic operators include:
  - + addition
  - - subtraction
  - * multiplication
  - / division (floating point and integer division depending upon type)
  - % modulo (remainder division)

Increment and decrement

- Adding 1 to a variable is so common, we have a custom operator for it
- We have three methods of incrementing (the same is true for decrement) variables in C
- Using addition (not technically an increment operator):

```
a = a + 1;
```

- Using the prefix increment operator

```
++a;
```

- Using the postfix increment operator

```
a++;
```

- What's the difference between these two statements?
  - A lot of the time, there is no difference in practice
  - **HOWEVER**
  - The semantics are technically different
  - How are they different?
    * ++a increments a and then returns the value of a (meaning the value returned has already been incremented)
    * a++ returns the value of a and then increments a (meaning the value returned has *not* yet been incremented)
  - When does this matter?
    * Two cases:

```
int a = 5;
int b = 5;
```

```
int c = ++a; // c has the value of 6 after this executes, a has the
    value of 6
int d = a++; // d has the value of 5 after this executes, a has the
    value of 6
```

- All of the above also applies for the decrement operator `--`, which subtracts 1 from a variable instead of adding 1

Floating point vs. Integer division

- We have a few different types of division in C++
- Noticing when we are using floating point vs. integer division is a common mistake
- Floating point division occurs when at least one of the operands is a floating point variable or floating point literal
- Integer division occurs when both operands are integers

```
int a = 5;
int b = 7;
double c = 5.0;
int int_div1 = a / b; // result is 0, because we are doing integer
    division and 5/7 is 0r5
int int_div2 = b / a; // result is 1, because we are doing integer
    division and 7/5 is 1r2
double float_div1 = c / b; // result is 0.71, because we are doing
    floating point division and 5.0/7 is 0.71
double float_div2 = a / b; // result is 0.0, because we are doing
    integer divion and 5/7 is 0r5
```

- The last line above may be surprising, but the right-hand side of the assignment performs integer division, resulting in `0`, and then the `0` is assigned to a double, resulting in `0.0`

Modulo (remainder division)

- Remember integer division from elementary school?
- e.g. `7/5` was `1r2` (1 with a remainder of 2) because 5 goes into 7 one time with a remainder of 2.
- When you divide two ints, you only get the quotient (number of times the denominator goes into the numerator).
- Modulo `%` gives us a way to get the remainder from the quotient division.
- Modulo is *extremely* useful.
  - It lets you add a bound to possible values.

- For instance, suppose you want to pick a random number between 0 and 9.
- Let's say you have a `rand()` function that returns a random number between 0 and a really, really big number (say 10000000000).
- You can do `rand()% 10` and you are guaranteed to get a number between 0 and 9.
- It doesn't matter how big the number is, the remainder *must* be between 0 and 9.
- Otherwise, the quotient increments

calculations.cpp

```
/* Let's try writing some calculations... */
#include <iostream>

using namespace std;

int main() {
   float value1 = 0.0, value2 = 0.0;
   int i1 = 0, i2 = 0;

   /* Prompt for values */
   cout << "\t\tCalculation Program\n\n";
   cout << "Please enter two values: ";
   cin >> value1;
   cin >> value2;

   /* perform calculations */
   cout << "Their sum: " << value1 + value2 << endl;
   cout << "Their product: " << value1 * value2 << endl;
   cout << "The first minus the second: " << value1 - value2 << endl;
   cout << "The second minus the first: " << value2 - value1 << endl;
   cout << "The first divided by the second: " << value1 / value2 <<
      endl;
   cout << "The second divided by the first: " << value2 / value1 <<
      endl;

   /* what if they are integers? */
   cout << "Let's try again, as if the values were int\n";
   i1 = (int) value1;
   i2 = (int) value2;
   cout << "Their sum: " << i1 + i2 << endl;
   cout << "Their product: " << i1 * i2 << endl;
   cout << "The first minus the second: " << i1 - i2 << endl;
```

```
    cout << "The second minus the first: " << i2 - i1 << endl;
    cout << "The first divided by the second: " << i1 / i2 << endl;

    /* FYI: The modulo operator is only defined on int operands... */
    cout << "The modulus of the first by the second: " << i1 % i2 <<
       endl;
    cout << "The second divided by the first: " << i2 / i1 << endl;
    cout << "The modulus of the second by the first: " << i2 % i1 <<
       endl;
    return( 0 );
  }
}
```

Conditionals

- There is no meaningful program that doesn't demonstrate some basic decision-making skills
- For instance, "I will continue driving through the intersection" is not a statement a human would act upon. But "I will stop if the light is red, go if the light is green, go only if I can safely pass if the light is yellow" might be a reasonable driving policy.
- A conditional tells the computer to only execute a block of code if a particular condition has been satisfied (i.e. that condition is true).
- `if...else` is the most common conditional statement
- `switch...case` are used as a shorthand version of `if...else`
- In C, logic is a form of arithmetic
  - `0` represents false
  - **any** other value represents true
  - Logical and arithmetic operators are treated as the same thing in C

Relational Expressions

- Consider the following relational and equivalence operations

```
a < b     // 1 if a is less than b, 0 otherwise
a > b     // 1 if a is greater than b, 0 otherwise
a <= b    // 1 if a is less than or equal to b, 0 otherwise
a >= b    // 1 if a is greater than or equal to b, 0 otherwise
a == b    // 1 if a is equal to b, 0 otherwise
a != b    // 1 if a is not equal to b, 0 otherwise
```

- Please remember = is for **assignment** and == is for **equality**

- The following conditionals are equivalent due to the definition of `true` and `false` in C++:
  - `false` is equivalent to `0` and `true` is equivalent to anything non-zero.
  - I would recommend avoiding comparing against any integer values and instead compare against `true` or `false` directly.

```
if (foo()) {
  // do something
}
if (foo() == true) {
  // do something
}
if (foo() != false) {
  // do something
}
if (foo() != 0) {
  // do something
}
```

selection.cpp

```
/* Let's try writing some conditional logic... */
#include <iostream>

using namespace std;

int main() {
    float value1 = 0.0, value2 = 0.0;
    /* Prompt for values */
    cout << "\t\tConditional Logic Program\n\n";
    cout << "Please enter two values: ";
    cin >> value1;
    cin >> value2;

    if (value1 < value2) {
        cout << "value1 is less than value2\n";
    }
    if (value1 > value2) {
        cout << "value1 is greater than value2\n";
    }
    if (value1 <= value2) {
        cout << "value1 is less than or equal value2\n";
```

```
    }
    if (value1 >= value2) {
        cout << "value1 is greater than or equal value2\n";
    }
    if (value1 == value2) {
        cout << "value1 equals value2\n";
    }
    if (value1 != value2) {
        cout << "value1 does not equals value2\n";
    }
    /* FYI: due to rounding errors, it is unreliable to test for
       equality on floating point numbers */
    return( 0 );
}
```

If-Else Statements

- Executes a block of code if particular conditions have been met
- Basic syntax:

```
if (/* condition goes here */) {
  /* if the condition is non-zero (true), this code will execute */
} else {
  /* if the condition is 0 (false), this code will execute */
}
```

- The first block executes if the condition is true, otherwise the second block executes
- The `else` is completely optional
- An `if` can directly follow an else, creating a chain of conditions to check:

```
if (a > b) {
  c = a;
} else if (b > a) {
  c = b;
} else {
  c = 0;
}
```

- This code sets the variable `c` equal to the greater of the two variables `a` or `b`, or `0` if `a` and `b` are equal.
- What's the point of the `else` when you can just do this:

```
if (a > b) {
  c = a;
}

if (a < b) {
  c = b;
}

if (a == b) {
  c = 0;
}
```

1. Multiple `if`'s could be true, there is no guaranteed mutual exclusion
2. Evaluating `if` statements takes time (since the condition must be checked).

- If you only have single statement to statement to execute in your `if` or `else`, you do not need to put a block.
  - **But you should.**
  - Consider the following

```
if(5 < 10)
  cout << "I am inside the if\n";
else
  cout << "I am inside the else\n";
  cout << "5 is not less than 10\n";
```

- Only the single statement following the `if` or `else` is associated with the conditional…so when will `5 is not less than 10` be printed?
  - **Every time**
  - Indentation cannot be trusted!
  - Here's a version with indentation reflecting the semantics of the program:

```
if(5 < 10)
  cout << "I am inside the if\n";
else
  cout << "I am inside the else\n";
cout << "5 is not less than 10\n";
```

- How do you fix this?
  - **Always use block statements!**

```
if(5 < 10) {
```

```
  printf("I am inside the if\n");
} else {
  printf("I am inside the else\n");
  printf("5 is not less than 10\n");
}
```

Looping

- Loops enable programmers to tell the computer to repeat a particular block of code multiple times.
  - Is is generally impractical to use conditionals a large number of times.
- Consider how a dishwasher might describe their time at work.
  - Unlikely the dishwasher would say "I watch a dish, and then another dish, and then another dish, ..."
  - More like they would say "I washed dishes the entire time I was at work"

While Loops

- A while loop is the most basic type of loop.
- **while** loops run until a specific controlling condition is not satisfied (i.e. false).
  - The controlling condition is checked *before* the loop executes and every time the loop loops.
- Syntax:

```
while(condition){
  //loop body
}
```

- Basic example:

```
int a = 1;
while (a < 100) {
  cout << "a is " << a << endl;
  a = a * 2;
}
```

- How many times will this loop execute?
  - 7, last time this executes a is set to 128 at the end of the loop
- **A critical note**: something must change in the loop such that the condition is eventually false and the loop exits

    - Otherwise, this is called an infinite loop.
- Consider the following:

```
int a = 1;
while (42) {
  a = a * 2;
}
```

- The controlling condition in the **while** never changes, and therefore will run forever (since 42 evaluates to true).
- **break** and **continue**
    - Allows you to control the flow of the loop from within the loop
    - **break** will immediately exit the loop
    - **continue** will skip the remainder of the block and start at the controlling conditional statement again.

```
int a = 1;
while (42) { // loops until the break statement in the loop is executed
   cout << "a is %d " << a << endl;
   a = a * 2;
   if (a > 100) {
     break;
   } else if (a == 64) {
     continue;  // Immediately restarts at while, skips next step
   }
   cout << "a is not 64\n";
}
```

- Similar to **if**, you may omit the braces for the block of code associated with the while loop
    - However, this is not recommended for the same reasons as with an **if** statement
    - Grouping of statements is potentially ambiguous (to the programmer, not the computer) that can lead to bugs

```
int a = 1;
while (a < 100)
  a = a * 2;
```

- This will just increase a until it is above 100
- When a loop ends, the program goes back to the while statement's controlling condition.
    - If the condition is true, the loop executes again
    - If the condition is false, the loop exits

  - The computer does *not* continuously check the controlling condition after each statement in the loop executes. It only checks at the end of every loop
  - If you need to end the loop during the middle of the loop's block, use a `break` to check for the necessary conditions

Do-While Loops

- The do-while loop is the same as a while loop, except the loop controlling condition is checked at the end of the loop rather than at the beginning
- Means the loop is guaranteed to execute at least one time.
- Syntax:

```
do {
  /* do stuff */
} while (condition);
```

- Note: he terminating `;` is required.
- `break` and `continue` operate the same as with other loops (the controlling condition will still be checked before executing the loop body again when using `continue`

Program style

- When writing your own programs, it's important to:
  1. Keep indentation clean and readable
  2. Use sensible variable names (don't use `a`, `b`, `c`, etc like I've been using in these small examples. Use names like `change` or `pounds` or `amount`)
  3. Write comments in your code. This will help yourself and future programmers that come across your code

Exercises

1. Write a C++ program to check whether a given number is even or odd

```
#include <iostream>

using namespace std;

void main()
{
  int num1, rem1;
```

```

  cout << "Input an integer : ";
  cin >> num1;
  rem1 = num1 % 2;
  if (rem1 == 0)
    cout << num1 << " is an even integer\n";
  else
    cout << num1 << " is an odd integer\n";
}
```

2. Write a C++ program to determine if a inputted integer is a palindrome

```
#include <iostream>

using namespace std;

int main()
{
  int n, num, digit, rev = 0;

  cout << "Enter a positive number: ";
  cin >> num;

  n = num;

  do
  {
    digit = num % 10;
    rev = (rev * 10) + digit;
    num = num / 10;
  } while (num != 0);

  cout << "The reverse of the number is: " << rev << endl;

  if (n == rev)
    cout << "The number is a palindrome\n";
  else
    cout << "The number is not a palindrome\n";

  return 0;
}
```