CSCI 240 Lecture Notes -- Part 1

A program = data + instructions

Data:

- there are several ***data types*** (numbers, characters, etc.)
- each individual data item must be ***declared*** and ***named***
- each individual data item must have a ***value*** before use
- *initial values* come from
  - program instructions
  - user input
  - disk files
- program instructions can alter these *values*
- original or newly computed *values* can go to
  - screen
  - printer
  - disk

Instructions:

- for data *input* (from keyboard, disk)
- for data *output* (to screen, printer, disk)
- *computation* of new values
- program control (decisions, repetition)
- modularization (putting a sequence of instructions into a package called a function)

The C++ Language - Introduction

C++ is a *superset* of the C programming language. Any C code is also C++ code, but some C++ code is not C. Much of what you learn in this course could also be used in a C program. There are lots of new features and capabilities in C++. We will learn some of them (but far from all of them) in this course.

The C++ Language is made up of

- keywords/reserved words (if, while, int, etc.)
- symbols: { } = | <= ! [ ] * & (and more)
- programmer-defined names for variables and functions

These programmer-defined names:

- 1 - 31 chars long; use letters, digits, _ (underscore)
- start with a letter or _
- are case-sensitive: *Num* is ***different*** than *num*
- should be meaningful: *studentCount* is better than *s* or *sc*

C++ Data Types

Each data item has a **type** and a **name** chosen by the programmer. The **type** determines the **range** of possible **values** it can hold as well as the **operations** that can be used on it. For example, you can add a number to a numeric data type, but you cannot add a number to someone's name. (What would "Joe" + 1 mean?)

Commonly-used data types

int (integer numbers)

- no decimal point, comma, or $
- leading + - allowed
- range (on our system): about plus or minus 2 billion
- int **constants** are written as: 1234 -3 43

To make a variable that your program can use, you must *declare* it.

Declaration Examples:

```
int x;            //gives type and name; no value yet

int x, y;         //declares 2 ints; no values yet

int population = 16000;   //declares & sets the initial value
```

So you could picture the results of the last two lines as:

x	y	population
??	??	16000

Note: It is **critically** important that variables have values before they are used in a program. In the examples above, x and y have no values even though they have been declared. More accurately, they do have values but those values could be anything between -2 billion and +2 billion. Before you **use** x or y you would have to give them values.

float and double (floating point or real numbers)

These data types are also commonly referred to as ***real*** variables (following mathematics usage) when it is not important which one we mean and we don't want to always have to say "float or double".

- has decimal point
- leading + - allowed
- no comma, $
- range (float) plus or minus 10 to 38th (limited to ~ 6 significant digits)
- range (double) plus or minus 10 to 308th (limited to ~12 significant digits)
- floating point constants are written as 3.08 -32.0 0.15 9.3e7 (9.3 times 10 to the 7th power)

Declaration Examples:

```
float  pi = 3.1416;   //declares, names, sets init value

double x = 3.5,       //note comma here
       y = 1245.6543; //can use > 6 digits

OR

double x   = 3.5;
double y   = 1245.6543;

float  big = 5.28e3;  // exponential notation: 5280
```

char (single characters)

- can hold just one character
- char constants are written with single quotes: 'a'

Declaration examples:

```
char ch;
char choice = 'q';
```

string (character "strings" or sequences)

- can hold 0 to many characters
- string constants are written with double quotes: "Hello, world"

Declaration Examples:

```
string s;

string MyName = "Amy";
```

Arithmetic Operations and Examples

The arithmetic operators are:

- + addition
- - subtraction or unary negation (-5)
- * multiplication
- / division (see special notes on division below)
- % modulus division -- integer remainder of integer division

Note: there is no exponentiation operator.

Special Note on Division:

Recall 3rd grade division:

```
    3  R1
   ______       4 is divisor
 4| 13          13 is dividend
    12          3 is quotient
     1          1 is remainder
```

In C++, a division with *2 int operands* has an *int* resulting value:

```
   13/4 --> 3   // int quotient
   13%4 --> 1   // int remainder
```

But with 1 or 2 float/double operands, the resulting value is a float or double:

```
So: 13.0 / 4 --> 3.25
    13 / 4.0 --> 3.25
```

Be aware. Forgetting this can easily cause an error.

Arithmetic Expressions are formed by *operands* and *operators*. Operands are the values used, operators are the things done to the numbers. Parts of arithmetic expressions are often grouped by () for clarity or to affect the meaning of the expression:

```
//declare and initialize some variables

int x = 11,
    y = 2;

double realnum = 2.0;
```

expression	value	notes
x + y	13	
x * y	22	
x * y + x	33	
x - y	9	
-x + y	-9	unary negation: "minus x"
x / y	5	int since both ops are int
x % y	1	rem when x divided by y
x / realnum	5.5	one op is real so result is real

Note: The expressions above *by themselves* are not valid C++ statements - they would be *part* of a statement. But each expression (if it is *in* a valid C++ statement) has a *value*.

Also - the spaces written between operands and operators are not required by C++, but do improve legibility.

More complex expressions are evaluated by C++ using **Rules of Precedence** (like algebra)

1. sub-expressions in ()
2. unary -
3. * and / - left to right
4. + and - - left to right

In the examples below, the red numbers show what operation is done as C++ evaluates and expression:

```
3 + 5 * 4 / 2 =
3 + 20 / 2 =
3 + 10 =
13
```

But () can be used to change this (or to make expression more readable)

```
(3 + 5) * 4 / 2 =
8 * 4 / 2 =
32 / 2 =
16
```

Assignment Statement/Operation

The symbol "=" is a **command**. It means: "*evaluate* the **expression** on the right and *store* this **value** in the **variable** on the left"

In general:

```
someVariable = some expression;
```

Examples:

```
x = y * 3;

x = x + y;
```

Notice that there is *always exactly one variable on the left* to receive the value of the expression on the right.

This is **NOT** like algebra; it is **NOT** an equation.

```
Algebra: x = 1    (T or F for a given x)
C++:     x = 1;   // store 1 into x


Algebra: x = x + 1   (false for all x)
C++:     x = x + 1; //find x+1; result in x
```

More Assignment Examples

```
//declare variables

double dAns, x, y;

int iAns, i, j;


//assign values via numeric literals

i = 5;
j = 8;
x = 4.0;
y = 1.5;


//assign values from complex expressions

i = i + 2;      // 7


dAns = x / y;   // 2.666..

iAns = j/2*2;   // 8/2*2 = 4*2 = 8

iAns = 8/5;          // 1

iAns = 8 % 5;   // 3

iAns = 5 / 8;   // 0

iAns = 5 % 8;   // 5

i = x/y;             // 2
```

The last needs an explanation. the value of the expression *x/y* is 2.666.. but *i* is an *int* variable and so cannot hold the .666 part. C++ simply drops it (truncates it) and so only the whole number part of the value is stored into *i*.

Notes:

1. All variables in expressions on the right must have defined values or you get random results. This is an example of what I meant before when I said variables must have values before they are used.

```
int x, y;       // declared; no init values

x = y + 1;      // ??? in x

int x;          // declared; no init value
int y = 2;      // declared; given a value

x = y + 1;      // x has value 3.
                // It is Ok if x has no value before this
```

Inadvertent use of un-initialized variables is one of the most common causes of program errors. Learn to be very careful about this.

2. You can do multiple assignments on one line:

```
x = y = z = 3;  //all get value 3
```

3. Naming Variables: use meaningful names that explain themselves to reader of the program. (You, me, the maintainer of the program after you go to your next job...)

```
No      Yes
s       score
cs      class_size      ClassSize       classSize
cav     class_average   ClassAvg        classAvg
num     student_count   NumStudents     numStudents
stnum   student_id      StudentID       studentID
```

Sample C++ program

```
#include <iostream>         // a.
#include <iomanip>

using namespace std;        // b.

int main()                  // c.
{
int num1, num2, sum;        // d.

num1 = 10;                  // e.
num2 = 15;

sum = num1 + num2;          // f.

return 0;                   // g.
}
```

Notes:

a. C++ standard libraries that must be included at the beginning of C++ programs

b. So the standard (std) objects can be used from the included libraries

c. program name is always *main* with *int* (which stands for integer) before it

d. *num1*, *num2* and *sum* are the data items (which we often call variables)

e. *num1* and *num2* get initial values from instructions via *assignment* (the = is the *assignment* command)

f. *sum* gets its value from an arithmetic expression

g. return an integer: *main* "promised" to provide an integer "answer". In this program (and usually) the "answer" doesn't matter. But we have to keep the promise, so we just return 0. We will discuss this further later.

h. on any line, typing // turns the rest of the line into a *comment*. A comment is not "seen" by the computer - it is not part of the program. It is used by human readers of the program to explain something in the program.

i. there are no program control or modularization statements

j. this program has no input or output

k. notice the semicolons - where they are and aren't

Output

We will use *cout* for output. *cout*'s task is to display information on the screen.

The simplest form of *cout* is:

```
cout << "Hello";
```

You can output (for display) several values, *including the values calculated and stored into variables*, by stringing them together connected by "<<":

```
cout << "The average is: " << avg;     // avg is a variable

cout << "The temperature is: "
     << temp
     << " and the rainfall today was: "
     << rainAmt;
```

Note that I have put each item to display on a separate line for clarity. This is both legal and is usually desirable if you have more than 2 values.

There are a couple of special techniques to position cursor before or after you display characters, using an **escape sequence**. (A \ followed by a char. For example, \n is the *newline* escape sequence.) Note that this is a **back**slash.

```
cout << "\nhello";    //moves to next line, shows hello

cout << "hello\n";    //shows hello, then goes to new line

cout << "hel\nlo";    //shows hel, then lo on next line
```

Note: to *display* a single \, you must code two: \\

```
cout << "the \\ char is special";
```

This shows: "the \ char is special"

Also, there is an alternate way to move the cursor to the beginning of a new line: the special value "endl":

```
cout << endl << "Some stuff" << endl;

cout << "\nSome stuff\n";
```

Both will first move the cursor to a new line, then display "Some stuff" and then move the cursor to the next line.

Note that if you don't include endl or \n, all your output will be on one line:

```
cout << "The average is: " << avg;       // avg is a variable

cout << "The temperature is: "
     << temp
     << " and the rainfall today was: "
     << rainAmt;
```

This will display something like this:

```
The average is 77.5The temperature is: 78 and the rainfall today was: 1.32362
```

Input

We will use *cin* for input. *cin*'s task is to accept user input and store it into variables.

Note that

- the C++ *language* itself has no defined I/O functions or commands; just a *standard library* that can accomplish I/O. We could use other libraries, including those from the C language or those developed by others.
- most often when we want user input, we will need to do two things:
  1. tell the user what to enter and then
  2. get the value entered.

So typically we will do something like this:

```
int i;                                // a.
...
cout << "Enter the temperature: ";    // b.
cin >> i;                             // c.
```

Notes:

a. we declare an int to hold the value to be typed in

b. we tell (*prompt*) the user to enter something

c. we get the value that is entered by the user

d. when a *cout* occurs in a program, the characters are displayed immediately on the screen

e. when *cin* occurs in a program, the program ***waits*** for user to type in a value and press ***<Enter>***.

f. the << and >> are meant to suggest the direction the data is going:
  - for *cout*, the prompt is going from the string constant through *cout* to the screen
  - for *cin* the data entered is coming in from the keyboard, through *cin*, to the variable i

g. for numeric input with *cin*, if the user types a non-number like "w" or "23e" then results are unpredictable. (For a double or float, "2e5" is ok - it's the same as 200000)

h. the variable receiving the value from *cin* must be declared (of course) but need not have an initial value. Any previous value in that variable is lost.

i. details about formatting numeric output (number of decimal places, etc.) will be covered later.

j. *cin* "understands" the different data types we have covered. So you can do *cin* with int, float, double, char, and string. But you **must** ensure that the data item getting the value is compatible with what the user is asked to enter. You can't store "jim" into an int, for example. You can store the value 4 into a double, however.

Program Error Types

1. Compile error - invalid C++ statements; so your code cannot be translated to machine language. Examples:

   1. integer x, y;
   2. num1 = num1 + num2

In example 1 you must use int, not integer. In example 2 the ; at the end of the line is missing

2. Link error: one of the *functions* used in the program is not found when it's time to run the program. In the program above, there are no such functions, but we could encounter this problem later.

3. Run-time error - program does not run to completion. Example:

- divide by zero: set x = 10 and y = 0 and then try to compute x/y

4. Logic error - program runs to completion, but the results are wrong. Example:

- you really wanted x - y but coded x + y
- in calculating an average, you use the wrong number to divide

Fixing 3 and 4 is called debugging.

You must learn to understand that the computer did **what you said**, not necessarily **what you meant**.