CS 101: Computer Programming and Utilization

Jul-Nov 2017

Umesh Bellur
(cs101@cse.iitb.ac.in)

Lecture 4: Variables, Data Types, and Expressions
Recap

• In the previous class, we learnt that computers essentially do arithmetic operations on numbers stored in the memory.
• We also looked into various number representations – decimal, binary, hex.
• AND how to store fractions in binary, negative or signed numbers in binary (sign magnitude and twos complement) and store very large and very small numbers using the IEEE FP representation on the machine.
• Now we will learn details of how different types of numbers are represented, stored and referred to in a program.
Outline

• Variables and their types
• Assignments and other expression types
• How to read numbers into the memory from the keyboard
• How to print numbers on the screen
• Many programs based on all this
Reserving Memory For Storing Numbers

• Before you store numbers in the computer's memory, you must explicitly reserve space for storing them in the memory.
• This is done by a variable declaration statement.
• variable: name given to the space you reserved.
• You must also state what kind of values will be stored in the variable: data type of the variable.

Byte#5 reserved for some variable named, \texttt{characterVar}, say.
Variable Declaration

• A general statement of the form:

\[
data\_type\quad variable\_name;\]

Creates and declares variables

• Example from polygon program:

\[
int\ sides;\]

• \textit{int} : name of the data type. Short form for integer. Says
• reserve space for storing integer values, positive or negative, of a standard size
• Standard size = 32 bits on most computers

• \textit{sides} : name given to the reserved space, or the variable created
```c
int sides;
```

Results in **memory** of size 32 bits being reserved for this variable. The program will refer to it by the name `sides`. **Sides** therefore refers to where the integer is stored.
Variable Names: Identifiers

- Sequence of one or more letters, digits and the underscore “_” character
- Should **NOT** begin with a digit
- Some words such as `int` cannot be used as variable names. *Reserved* by C/C++ for its own use
- **Case Matter:** ABC and abc are distinct identifiers

- **Valid Identifiers:** sides, telephone_number, x, x123, third_cousin

- **Invalid identifiers:** #sides, 3rd_cousin, third cousin
- Recommendation: use meaningful names, describing the purpose for which the variable will be used
Some Other Data Types Of C++

- **unsigned int**: Used for storing integers which will always be positive
  - 1 word (32 bits) will be allocated
  - Ordinary binary representation will be used
- **char**: Used for storing characters or small integers
  - 1 byte will be allocated
  - ASCII code of characters is stored
- **float**: Used for storing real numbers
  - 1 word will be allocated
  - IEEE FP representation, 8 bits exponent, 23 bits significand
- **double**: Used for storing real numbers
  - 2 words will be allocated
  - IEEE FP representation, 11 bits exponent, 52 bits significand
Types of variables

int a;      // Signed Integer 4 bytes
long int b;  // Signed integer 4 or 8 bytes
short int s;  // Signed integer 2 bytes
char c;      // Signed integer 1 byte
unsigned int d;  // Unsigned int 4 bytes
unsigned long e;  // Unsigned int 4 or 8 bytes
unsigned short f;  // Unsigned int 2 bytes
unsigned char g;  // Unsigned int 1 byte
float f;      // Note that floats cannot be unsigned.
double dd;    // double cannot be unsigned, 8 bytes
long long int; // 16 bytes but may not be available
long double ddd; // double precFloating Point 16 bytes,
Boolean and Int

• There is no Boolean type
• 0 is False and anything other than 0 is True.

```c
if(5){
    //always executed
}
if(0){
    //never executed
}
```
Variable Declarations

• Okay to define several variables in same statement
  – float mass, accel;

• The keyword long says, I need to store bigger numbers, so give me more than the usual space.
  – long unsigned int crypto_password;
    • long unsigned int: Likely 64 bits will be allocated
  – Long double more_precise_value;
    • long double: likely 96 bits will be allocated
Variable *Initialization*

• **Initialization** – an INITIAL value is assigned to the variable

  *the value stored in the variable at the time of its creation*

• Variables *i*, *vx*, *vy* are declared AND are initialized
  – 2.0e5 is how we write 2.0*10^5
  – ‘f’ is a character constant representing the ASCII value of the quoted character
  – *result* and *weight* are declared but not initialized

```
int i=0, result;
float vx=1.0, vy=2.0e5, weight;
char value = 'f';
```
const double pi = 3.14;

The keyword `const` means: *value assigned once cannot be changed in the program.*

Useful in readability AND uniformity of a program

\[
\text{area} = \pi \times \text{radius} \times \text{radius};
\]

reads better than

\[
\text{area} = 3.14 \times \text{radius} \times \text{radius};
\]
Reading Values Into Variables from cin

- `>>` is a reading operator defined on `cin`

- Can read into several variables one after another

- If you read into a char type variable, the ASCII code of the typed character gets stored

- If you input the character ‘f’, the ASCII value of ‘f’ will get stored

```cpp
cin >> no_of_sides;
cin >> vx >> vy;
char command;
cin >> command;
```
Reading Values Into Variables (2)

- User expected to type in values consistent with the type of the variable into which it is to be read

- **Whitespaces** (i.e. space characters, tabs, newlines) typed by the user are ignored.

- **newline/enter key must be pressed after values are typed**
Printing Variables On The Screen

- Output stream `cout` defines a operator `<<`

- General form: `cout << variable;`
- Many values can be printed one after another
- To print newline, use `endl`

- Verbatim text can be printed by enclosing it in quotes. This one prints the text `Position:`, then `x` and `y` with a comma between them and a newline after them

- If you print a `char` variable, then the content is interpreted as an ASCII code, and the corresponding character is printed.
  
  G will be printed.

```cpp
cout << x;
cout << x << y;
cout << "Position:"
<< x << ", " <<
y << endl;

char var = 'G';
cout << var;
```
Statements
An Assignment Statement

Used to assign a value to a variable or store results of computation into a variable. Form: `variable_name = expression;`

```
name = "mynname";
s = u*t + 0.5 * a * t * t;
n = (3/x)*y + (45 * z);
```

**Expression**: can specify a formula involving constants or variables, almost as in algebra

- If variables are specified, their *values* are used.
- **operators** must be written explicitly
- multiplication, division have higher precedence than addition, subtraction
- multiplication, division have same precedence
- addition, subtraction have same precedence
- operators of same precedence will be evaluated left to right.
- Parentheses can be used with usual meaning
int x=2, y=3, p=4, q=5, r, s, t;

x = r*s;  // disaster. r, s undefined
r = x*y + p*q;
  // r becomes 2*3 + 4*5 = 26
s = x*(y+p)*q;
  // s becomes 2*(3+4)*5 = 70
t = x – y + p – q;
  // equal precedence,
  // so evaluated left to right,
  // t becomes (((2-3)+4)-5 = -2
Arithmetic Between Different Types Allowed

```c
int x=2, y=3, z, w;
float q=3.14, r, s;
r = x; // representation changed
    // 2 stored as a float in r  "2.0"

z = q; // store with truncation
    // z takes integer value 3

s = x*q; // convert to same type,
    // then multiply
    // Which type?
HIGHER PRECISION.
```
Evaluating varA op varB
e.g. x*q

- if varA, varB have the same data type: the result will have same data type
- if varA, varB have different data types: the result will have more expressive data type
- int/short/unsigned int are less expressive than float/double
  - shorter types are less expressive than longer types
Storing numbers of one type into variable of another type

C++ does the “best possible”.

```c
int x; float y;
x = 2.5;
y = 123456789;
```

x will become 2, since it can hold only integers. Fractional part is dropped.

123456789 is stored as 1.234567E8 (May have loss of precision!)
Integer Division

```cpp
int x=2, y=3, p=4, q=5, u;
u = x/y + p/q;
Cout << u << endl;
cout << p/y << endl;
```

- **x/y**: both are `int`. So truncation. Hence 0
- **p/q**: similarly 0 =&gt; u = 0;
- **p/y**: 4/3 after truncation will be 1
- So the output is 0 followed by 1
More Examples of Division

```c
int no_of_sides=100, i_angle1, i_angle2;
i_angle1 = 360/no_of_sides + 0.45; // 3
i_angle2 = 360.0/no_of_sides + 0.45; // 4

float f_angle1, f_angle2;
f_angle1 = 360/no_of_sides + 0.1; // 3.1
f_angle2 = 360.0/no_of_sides + 0.1 // 3.7
```
An Example Limited Precision

float w, y=1.5, avogadro=6.022E23;
w = y + avogadro;

- Actual sum : 602200000000000000000001.5
- $y + avogadro$ will have type float, i.e. about 7 digits of precision.
- With 7 digits of precision ($2^{23}$), all digits after the 7\textsuperscript{th} will get truncated and the value of $avogadro$ will be the same as the value of $y + avogadro$
- $w$ will be equal to $avogadro$
- No effect of addition!
Program Example

main_program{
    double centigrade, fahrenheit;
    cout << "Give temperature in Centigrade: ";
    cin >> centigrade;
    fahrenheit = centigrade * 9 / 5 + 32;
    cout << "In Fahrenheit: " << fahrenheit << endl; // newline
}

Prompting for input is meaningless in Prutor because it is non-interactive
Re-Assignment

- Same variable can be assigned a value again
- When a variable appears in a statement, its value at the time of the execution of the statement gets used

```cpp
int p=3, q=4, r;
r = p + q; // 7 stored into r
cout << r << endl; // 7 printed as the value of r
r = p * q; // 12 stored into r
cout << r << endl; // 12 printed as the value of r
```
In C++ "=" is assignment not "equal"

```cpp
int p=12;
p = p+1;
```

See it as:  
\[ p \leftarrow p+1; \quad \text{// Let } p \text{ become } p+1 \]

Rule for evaluation:

- FIRST evaluate the RHS and THEN store the result into the LHS variable
- So 1 is added to 12, the value of \( p \)
- The result, 13, is then stored in \( p \)
- Thus \( p \) finally becomes 13

\[ p = p + 1 \] is nonsensical in mathematics

“=” in C++ is different from “=” in mathematics
Repeat And Reassignment

main_program{
    int i=1;
    repeat(10){
        cout << i << endl;
        i = i + 1;
    }
}

This program will print 1, 2,…, 10 on separate lines
Another Idiom: Accumulation

```cpp
main_program{
    int term, s = 0;
    repeat(10){
        cin >> term;
        s = s + term;
    }
    cout << s << endl;
}
```

- Values read are *accumulated* into `s`
- Accumulation happens here using `+`
- We could use other operators too
Fundamental idiom

Sequence generation

• Can you make $i$ take values 1, 3, 5, 7, …?
• Can you make $i$ take values 1, 2, 4, 8, 16, …?

```cpp
int i=1;
repeat(10){
    cout << i << endl;
    i = i + 2;
}

int output=0; i = 0;
repeat(10){
    output = pow(2,i);
    cout << output << endl;
    i = i + 1;
}
```
Composing The Two Idioms

Write a program to calculate \( n! \) given \( n \).

```cpp
main_program{
    int n, nfac=1, i=1;
    cin >> n;
    repeat(n){
        nfac = nfac * i;
        i = i + 1;
    }
    cout << nfac << endl;
}
```

Accummulation idiom

Sequence idiom
Finding Remainder

• $x \% y$ computes the remainder of dividing $x$ by $y$
• Both $x$ and $y$ must be integer expressions
• Example

```c
int n=12345678, d0, d1;
d0 = n % 10;
d1 = (n / 10) % 10;
```

d0 will equal 8 (the least significant digit of $n$)
d1 will equal 7 (the second least significant digit of $n$)
Some Additional Operators

• The fragment \( i = i + 1 \) is required very frequently, and so can be abbreviated as \( i++ \)

  \( ++ \): increment operator. Unary

• Similarly we may write \( j-- \) which means \( j = j - 1 \)

  \( -- \): decrement operator. Unary
Intricacies Of ++ and --

++ and -- can be written after or before the variable. Both cause the variable to increment or decrement but with subtle differences

```cpp
int i=5, j=5, r, s;
r = ++i;
s = j++;
```

`i, j both` become 6 but `r` is 6 and `s` is 5.

++ and -- can be put inside expressions but not recommended in good programming
Compound Assignment

The fragments of the form `sum = sum + expression` occur frequently, and hence they can be shortened to `sum += expression`.

Likewise you may have `*=` , `-=` , `/=` …

```c
int x=5, y=6, z=7, w=8;
x += z; // x becomes x+z = 12
y *= z+w; // y becomes y*(z+w) = 90
```
## Arithmetic Operators

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>x=3+2; /<em>constants</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y+z; /<em>variables</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x+y+2; /<em>both</em>/</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>3−2; /<em>constants</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int x=y−z; /<em>variables</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y−2−z; /<em>both</em>/</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>int x=3∗2; /<em>constants</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int x=y∗z; /<em>variables</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x∗y∗2; /<em>both</em>/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>division</td>
<td>float x=3/2; /*produces x=1 (int /) */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>float x=3.0/2 /*produces x=1.5 (float /) */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int x=3.0/2; /<em>produces x=1 (int conversion)</em>/</td>
</tr>
<tr>
<td>%</td>
<td>modulus (remainder)</td>
<td>int x=3%2; /<em>produces x=1</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int y=7;int x=y%4; /<em>produces 3</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int y=7;int x=y%10; /<em>produces 7</em>/</td>
</tr>
</tbody>
</table>

Note the type conversions
Relational Operators (boolean result)

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>3&gt;2; /*evaluates to 1 */ 2.99&gt;3 /*evaluates to 0 */</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>3&gt;=3; /*evaluates to 1 */ 2.99&gt;=3 /*evaluates to 0 */</td>
</tr>
<tr>
<td>&lt;</td>
<td>lesser than</td>
<td>3&lt;3; /*evaluates to 0 */ ‘A’&lt;’B’ /<em>evaluates to 1</em>/</td>
</tr>
<tr>
<td>&lt;=</td>
<td>lesser than or equal to</td>
<td>3&lt;=3; /*evaluates to 1 */ 3.99&lt;3 /*evaluates to 0 */</td>
</tr>
</tbody>
</table>

int x = (3 >2); is equivalent to int x = 1;
int x = (3 < 3) is equivalent to int x = 0;
Equality and Inequality

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
<td>3==3; /*evaluates to 1 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'A'=='a'; /*evaluates to 0 */</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>3!=3; /*evaluates to 0 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.99!=3; /*evaluates to 1 */</td>
</tr>
</tbody>
</table>

Note that the "==" equality operator is different from the "+=" assignment operator.

Note that the "==" operator on float variables is tricky because of finite precision.
## Logical Operators

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
</table>
| &&       | AND     | `((9/3)==3) && (2*3==6); /*evaluates to 1 */`  
          |         | `('A' =='a') && (3==3) /*evaluates to 0 */`  |
| ||       | OR      | `2==3 || 'A' =='A'; /*evaluates to 1 */`  
          |         | `2.99>=3 || 0 /*evaluates to 0 */`  |
| !        | NOT     | `!(3==3); /*evaluates to 0 */`  
          |         | `!(2.99>=3) /*evaluates to 1 */`  |
**Bit Operations: Left and Right Shift `<<` `>>`**

- **`x >> i`**
  - Shifts bits of a number `x` to the right `i` positions
- **`x << i`**
  - Shifts bits of a number `x` to the left `i` positions
- **Example:**
  ```c
  int i, j;
  i = 5; // In binary i is 00000101
  j = (5 << 3); // In binary j is 00101000
  printf("i=%d j=%d\n"); // Output: i=5 j=40
  ```
- `x<<i` is equivalent to `x*2^i`
- `x>>i` is equivalent to `x/2^i`
Bitwise Operations: OR

- The "|" operator executes "OR" bit operation.

```c
unsigned x = 0x05; // 00000101
unsigned y = (x | 0x2);
// 00000101 | 00000010=00000111
```
Bitwise Operations: &

• The “&” operator executes “AND” bit operation.

```c
unsigned x = 5; // 00000101
unsigned y = (x & 0x3); // 00000101 & 00000011 = 0000001
```
Bitwise Operations: XOR ^

• The “^” operator executes “XOR” bit operation.
• XOR: 0^0==0, 0 ^1 == 1, 1^0==1, 1^1==0

unsigned x = 0x05; // 00000101
unsigned y = (x ^ 0x3);// 00000101 ^ 00000011 =00000110

Example:
// Inverting the bits of a number
unsigned x = 0x05; // 00000000 00000000 00000000 00000101
unsigned y = x ^ 0xFFFFFFFF;
//0xFFFFFFFF=11111111 11111111 11111111 11111111
    // y = 11111111 11111111 11111111 11111010
Bitwise Operations: NOT ~

• The "~" negates bits.

unsigned x = 0x05; // 00000000 00000000 00000000 00000101
unsigned y = ~x;
// ~00000101 = 11111111 11111111 11111111 11111010
Blocks and Scope

- Code inside {} is called a block.
- Blocks are associated with repeats, but you may create them otherwise too.
- You may declare variables inside any block.
- New summing program:
  - The variable term is defined close to where it is used, rather than at the beginning. This makes the program more readable.
  - But the execution of this code is a bit involved.

```cpp
// The summing program
// written differently.

main_program{
    int s = 0;
    repeat(10){
        int term;
        cin >> term;
        s = s + term;
    }
    cout << s << term << endl;
}
```
How definitions in a block execute

• Basic rules
• A variable is defined/created every time control reaches the definition.
• All variables defined in a block are destroyed every time control reaches the end of the block.
• “Creating” a variable is only notional; the compiler simply starts using that region of memory from then on.
• Likewise “destroying” a variable is notional.
• New summing program executes exactly like the old, it just reads different (better!).
Shadowing and scope

- Variables defined outside a block can be used inside the block, if no variable of the same name is defined inside the block.
- If a variable of the same name is defined, then from the point of definition to the end of the block, the newly defined variable gets used.
- The new variable is said to "shadow" the old variable.
- The region of the program where a variable defined in a particular definition can be used is said to be the scope of the definition.
Example

main_program{
    int x=5;
    cout << x << endl; // prints 5
    {
        cout << x << endl; // prints 5
        int x = 10;
        x *= 2;
        cout << x << endl; // prints 20
    }
    cout << x << endl; // prints 5
}
Concluding Remarks

Variables are regions of memory which can store values.

Variables have a type, as decided at the time of creation.

Choose variable names to fit the purpose for which the variable is defined.

The name of the variable may refer to the region of memory (if the name appears on the left hand side of an assignment), or its value (if the name appears on the right hand side of an assignment).
Further Remarks

Expressions in C++ are similar to those in mathematics, except that values may get converted from integer to real or vice versa and truncation might happen. Truncation may also happen when values get stored into a variable. Sequence generation and accumulation are very common idioms. Increment/decrement operators and compound assignment operators also are commonly used (they are not found in mathematics).
More Remarks

Variables can be defined inside any block.

Variables defined outside a block may get shadowed by variables defined inside.
SAFE quiz

• What is the result of evaluating the expression (3+2)/4?
• What is printed by this code snippet: "float f=6.022E23; float r=f+2-f; cout<<r;"?
• What is printed by this code snippet: "int t=10; repeat(2){t=t-1.2;} cout<<t;"?
• What is printed by this code: "int i=2, j=3, k=4; i=j; j=k; k=i; cout << (i*j*k)"?