# Assignment statement: storing a value into a variable defined earlier

- Statement form: ***variable = expression;***

***s = u*t + 0.5 * a * t * t;***

- Expression : formula involving constants or variables, almost as in mathematics.
- Execution: The value of ***expression*** is calculated and stored into the variable which appears on the left hand side of the = symbol.
  - If ***expression*** contains variables they must have been assigned a value earlier – this value will be used to calculate the value of the expression.
  - If ***u, a, t*** have values 1, 2, 3 respectively, 1*3 + 0.5 * 2 * 3 * 3 = 12 will be stored in ***s***.
  - The value present in ***s*** before the execution of the statement is lost or "overwritten".
  - The values present in other variables including u, a, t does not change when the above statement executes.

# Rules regarding expressions

- Multiplication must be written explicitly.
  ***s = u t + 0.5 a t t;  // not legal***
- 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.
- Spaces can be put between operators and values as you like

s=u*t+0.5*a*t*t;  s = u*t + 0.5*a*t*t;  // both OK

# Examples

```
int x=2, y=3, p=4, q=5, r, s, t, u;
r = x*y + p*q;
     // 2*3 + 4*5 = 26
s = x*(y+p)*q;
     // 2*(3+4)*5 = 70
t = x — y + p — q;
     // ((2-3)+4)-5 = -2
u = x + w;
//wrong if w not defined earlier
```

# Example involving division

```
int x=2, y=3, z=4, u;
u = x/y + z/y;
```

What will this do?

- If dividend and divisor are both integers, then result is the quotient, i.e. remainder is ignored.
- u becomes: 2/3 + 4/3 = 0 + 1 = 1
- Program will give error if divisor is 0.

# Rules for storing numbers of one type into variable of another type

- C++ does the “best possible”.

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

- x will become 2, since it can hold only integers. Fractional part is dropped.
- 123456789 cannot be precisely represented in 24 bits, so something like 1.234567 e 9 will get stored.

# Evaluating "***A op B***" when A, B have different types

- If ***A,B*** have different data types: then they will be converted to "more expressive" data type among the two:
  - ***int/short/unsigned int*** are less expressive than ***float/double***
  - shorter types are less expressive than longer.
- The operation will then be performed, and the result will have the more expressive type.

# Examples

```
int nsides=100,

int iangle1,iangle2;
iangle1 = 360/nsides;
iangle2 = 360.0/nsides;

float fangle1, fangle2;
fangle1 = 360/nsides;
fangle2 = 360.0/nsides;
```

- 360/nsides : both are integer so integer division is used.
- Result 360/100 = 3
- 3 is stored in iangle1
- 360.0/nsides : numerator is double, so denominator also converted to double.
- Result 360.0/100.0 = 3.6
- But iangle2 is int, so integer part, 3, stored.
- 360/nsides evaluates to 3, stored in fangle1
- 360.0/nsides evaluates to 3.6, stored in fangle2

# Implication of limited precision

- ***float w,y=1.5, avogadro=6.022e23;***
- ***w = y + avogadro;***

- "Actual sum" : 602200000000000000000001.5
- Sum will have to be stored in variable ***w*** of type float.
  - But ***w*** can only accommodate 23 bits, or about 7 digits.
  - Thus all digits after the $7^{th}$ will get truncated.
  - To 7 digits of precision ***avogadro*** is same as ***y+avogadro***.
- ***w*** will receive the truncated value, i.e. ***avogadro***.
- This addition has no effect!

# 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.
}
// Do we need to write 9.0?
```

# Demo

# Where expressions can appear

- Expressions can appear on the right had side of an assignment.
- As a rule: expressions can appear wherever a plain number can

```
double u=1, t=2, a=3, v = u+a*t, s = (u+v)*t/2;
```

  – Initialization happens left to right.

```
int x=5*y, y=2; // incorrect

cout << u+a*t; // OK if u, a, t have values
forward(u*10); // forward 10 pixels
```

# A useful operator: % for finding remainder

- ***x % y*** evaluates to the remainder when ***x*** is divided by ***y***. ***x, y*** must be integer expressions.
- Example

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

- ***d0*** will equal the least significant digit of ***n***, 8.
- ***d1*** will equal the second least significant digit of ***n***, 7.

# Exercise

What are the results of evaluating the following expressions

- (1+2)/4
- (1.0+2)/4

Write a program that reads length in inches and prints it as x yards, y feet, z inches.

- Note that one yard is three feet, one foot (singular of feet) is 12 inches.
- You will find it useful to remember the properties of integer division and the % operator.

# What we discussed

- The assignment statement
- Rules about how arithmetic happens when an expression contains numbers of different types.
- Also to be noted is that real numbers are represented only to a fixed number of digits of precision in the significand, and so adding very small values to very large values may have no effect.
- What we did not discuss but you should read from the book: overflow, representation of infinity.

# Re assignment

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

```
int p=3, q=4, r;
r = p + q;          // 7 stored into r
cout << r << endl;  // 7 printed
r = p * q;          // 12 stored into r
cout << r << endl;  // 12 printed
```

# An interesting assignment expression

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

- Follow usual rule for evaluation:
  - First evaluate the value on the left hand side.
  - 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 math.

# Repeat and reassignment

# Repeat and reassignment

```
main_program{
  turtleSim();
  int i=1;
  repeat(10){
    forward(i*10);
    right(90);
    cout << i << endl;
    i = i + 1;
  }
  wait(5);
}
```

- First iteration:
  - 1 printed, i changes to 2.
- Second iteration:
  - 2 printed, 2 changes to 3.
- ...
- 10th iteration:
  - 10 printed, i changes to 11.
- Fundamental idiom: sequence generation.
- What does this draw?
  - “Rectangular spiral”

# Demo

# Other sequences can also be generated

- Can you make i take values 1, 3, 5, 7, …?
- Can you make i take values 1, 2, 4, 8, 16, …?
- Both can be done by making slight modifications to previous program.

# Another idiom: accumulation

```
// Program to read 10 numbers and add them.
main_program{
      int term, s = 0;
    repeat(10){
        cin >> term;
        s = s + term;
    }
    cout << s << endl;
}
// values read get "accumulated" into s
// Accumulation happens here using +
// We could use other operators too.
```

# Composing the two idioms: program to calculate n!

```
main_program{
  int i=1, n;   cin >> n;
  int nfac=1;
  repeat(n){
     nfac = nfac * i; // multiplied by 1 to n
     i = i + 1; // i goes from 1 to n
  }
  cout << nfac << endl; // n! printed
}
```

# Some additional operators

- The fragment "**i = i + 1**" appears very frequently, and so can be abbreviated as "**i++**".
- ++ : increment operator. Unary
- Similarly we may write "**j--**" which means "**j = j — 1**"
- -- : decrement operator

# Intricacies of ++ and --

- ++ and -- can be written after the variable, and this also cause the variable to increment or decrement.
- Turns out that expressions such as k = ++i; and k = i++; are legal in C++ and produce different results.
- Such assignments are described in the book for completeness.
- But they are somewhat hard to read, and so it is recommended you do not use them.
- Similarly with --.

# Compound assignment

- The fragments of the form “sum = sum + expression;” occur frequently, and so C++ allows them to be shortened to “sum += expression;”
- Likewise you may have *=, -=, …
- Example

```
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
```

- Note: **z, w** do not change.

# Exercise

- What does the following program do?

```
unsigned int n=7589, m=0;
repeat(5){
  m = 10*m + (n % 10);
  n = n/10;
}
```

- What are the values of the variables after the following statements execute

```
int i=1,j=2,k=3;
i=j;
j=k;
k=i;
k++;
i--;
```

# What we discussed

- The value of a variable can be changed (reassigned).
- Assignments such as i = i+1 or j = j*2 are allowed.
  - They are useful for generating sequences
  - Once you can generate sequences, you can compute expressions such as n!
  - Other uses will be seen later
- Because assignments such as i = i+1 and j = j*2 are very frequently needed, operators such as ++ and *= have been provided.

# 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.

```
// The summing program
// written differently

main_program{
      int s = 0;
  repeat(10){
    int term;
    cin >> term;
    s = s + term;
  }
  cout << s << 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!).

# Scope and Shadowing

- 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.
- Why do we care:
  - In a single English language document you might write "Let x denote" in several places, with the understanding that the x on page 5 is different from the x on page 37.
  - If you do not have an intervening "Let x denote .." the two xs might be same.
  - Something similar happens while writing large programs

# Example

```
main_program{
  int x=5;
  cout << x << endl;   // prints 5
  {
    cout << x << endl; // prints 5
    int x = 10;
    cout << x << endl; // prints 10
  }
  cout << x << endl; // prints 5
}
```

What if ***int x = 10;*** was ***x = 10;*** ?

# 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).

# More 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.

# More remarks

- Variables can be defined inside any block.
- Variables defined outside a block may get shadowed by variables defined inside.