Ch. 3: Variables, Data Types and expressions

CS 101 Computer Programming and Utilization
Varsha Apte
Slides based on Abhiram G. Ranade's slides

Recall

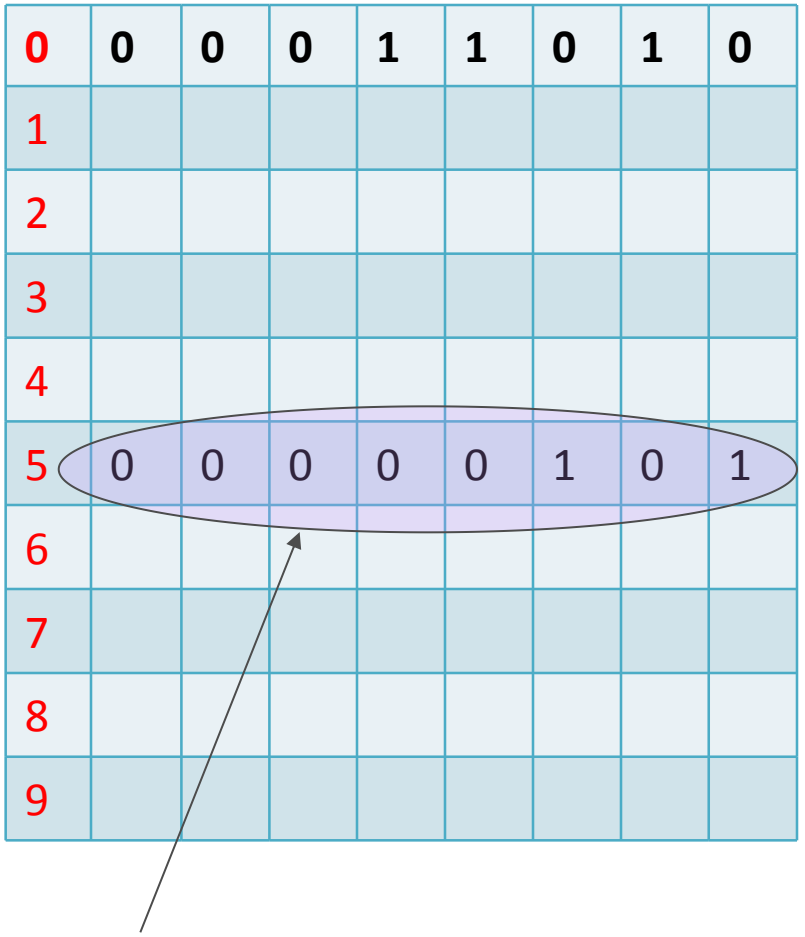
- In the previous slide set, we learnt that computers essentially do arithmetic operations on numbers stored in memory
- Now we will learn details of how different types of numbers are represented and stored, and refered to in a program

Outline

- How to store numbers in the memory of a computer.
- How to perform arithmetic.
- How to read numbers into 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 definition" 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.



Variable "definition"

General Statement form:

data_type_name variable_name;

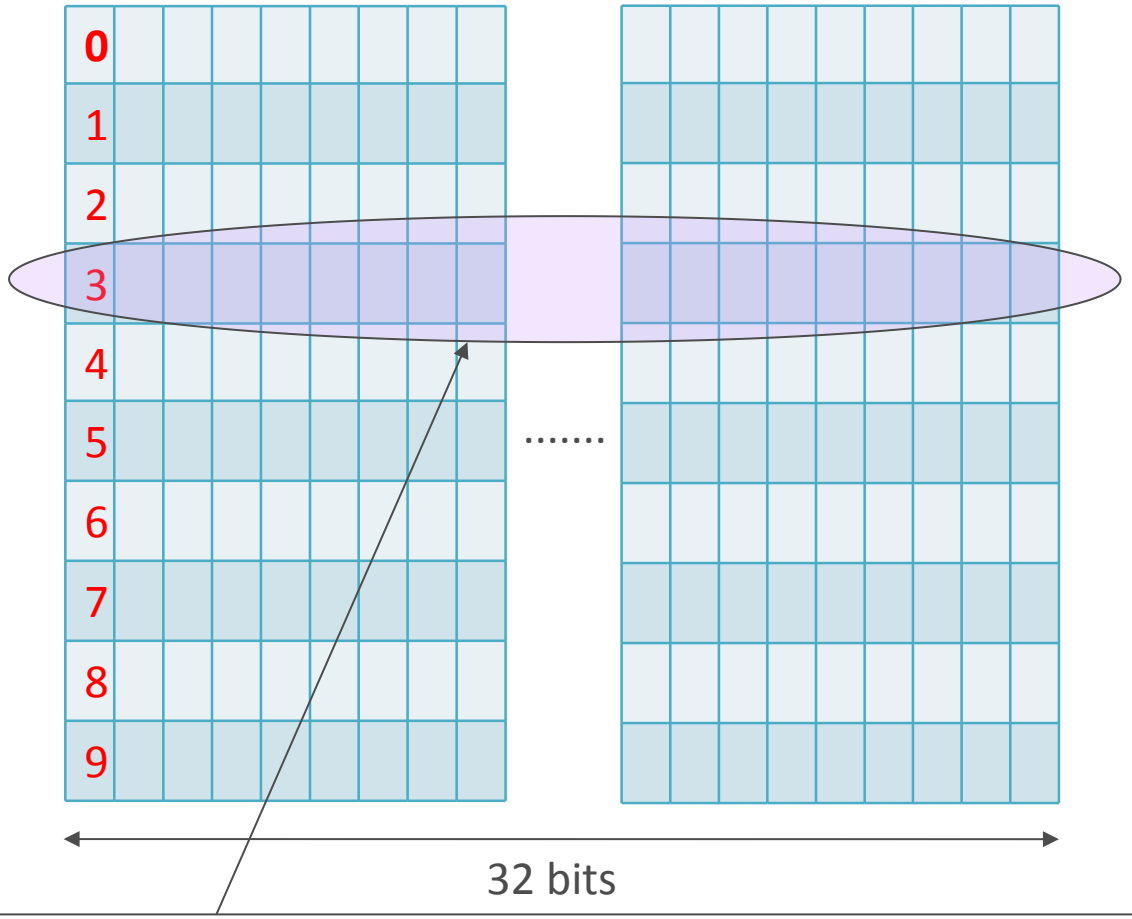
- Creates and defines the variables
- Example from chapter 1:

```
int nsides;
```

- **int** : name of the data type. Short for integer. Says
  - Reserve space for storing integer values, positive or negative, of a "standard" size.
  - Standard size = 32 bits on most computers.

- **nsides** : name given to reserved space, or the created variable.

Variable "definition"



int nsides;

Results in a "memory location" of size 32 bits being reserved for this variable. The program will refer to it by the name "nsides".

Variable names: “Identifiers”

- Sequence of 1 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++ for its own use.
  - case matters. ABC and abc are distinct identifiers
- Examples: nsides, telephone_number, x, x123, third_cousin
- WRONG identifiers: #sides, 3rd_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, 24 bits significand.
- double : used for storing real numbers
  - 2 words will be allocated.
  - IEEE FP representation, 11 bits exponent, 53 bits significand.

Variable definition

- OK to define several variables in same statement.

- Keyword long : says, “I need to store bigger or more precise numbers, so give me more than usual space.”

- long unsigned int: Likely 64 bits will be allocated.

- long double: likely 96 bits will be allocated.

```
//Examples
unsigned int telephone_number;
float velocity;
float mass, acceleration;

long unsigned int
    cryptographic_password;

long double
    more_precise_acceleration;
```

Variable initialization

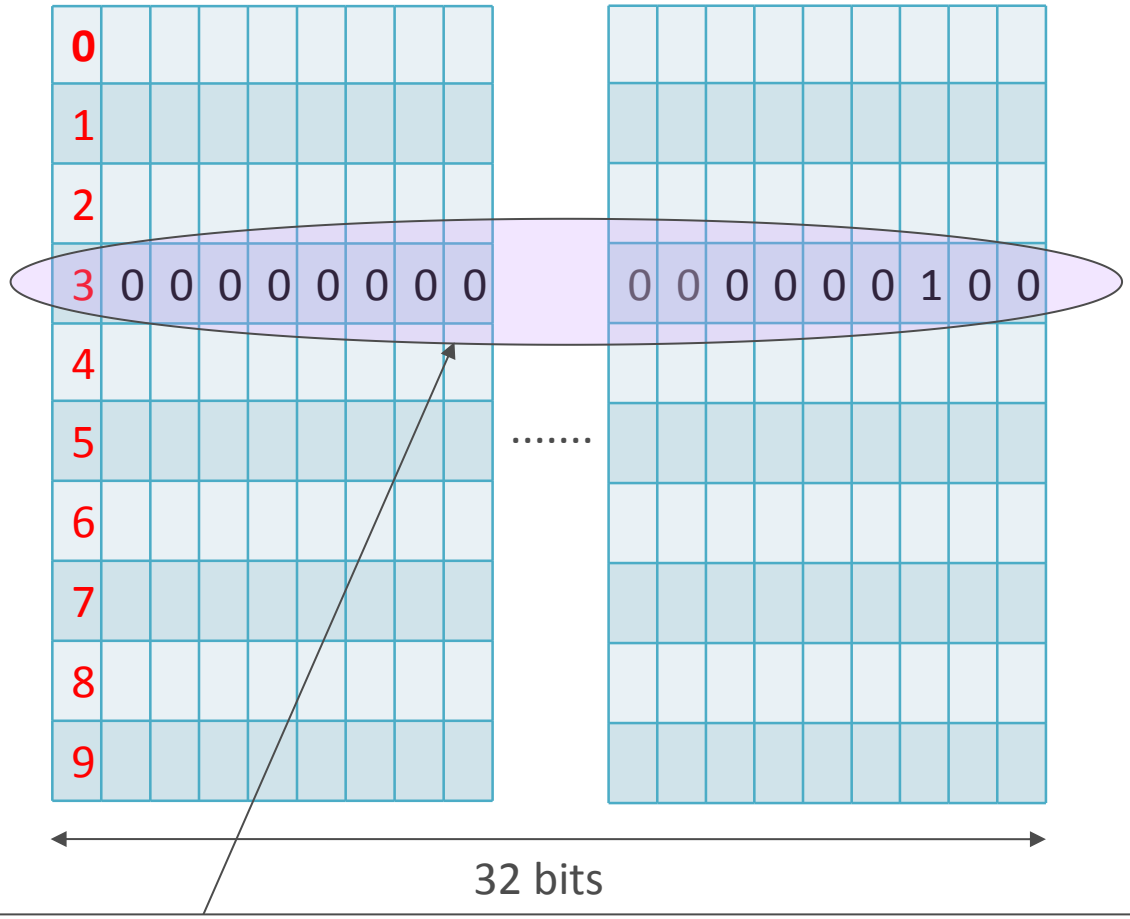
- 'Initialization" - the FIRST value a variable is assigned.
- A value can be stored in a variable at the time of creation
  - i, vx,vy given values as well as defined.
  - 2.0e5 is how you write $2.0*10^5$
  - ‘f’ is a “character constant”. It represents the ASCII value of the quoted character.
  - result and weight not initialized here

```
int i=0, result;

float vx=1.0, vy=2.0e5, weight;

char command = ‘f’;
```

Variable initialization



int nsides=4;

Results in a "memory location" of size 32 bits being reserved for this variable. The program will refer to it by the name "nsides".

Const keyword

```
const double pi= 3.14;
```

- The keyword const means : value assigned cannot be changed.
- Useful in readability of a program

```
area = pi * radius * radius*
```

reads better than

```
area = 3.14 * radius * radius
```

Reading values into variables

- We did this in chapter 1:

```
cin >> nsides;
```

- Can read into several variables one after another, like this:

```
cin >> vx >> vy;
```

- If you read into a *char* type variable, the ASCII code of the typed character gets stored.
  - If you type the character 'f', the ASCII value of 'f' will get stored.

```
char command;
cin >> command;
```

Some rules:

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

"Whitespace" = 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

General form: *cout << variable_name;*

- Many can be printed one after another.
- To print newline, use endl.
- Additional text can be printed by enclosing it in quotes.

This one prints the text “Position: “, then x, y with a comma will 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.

```
//Examples
cout << x;

cout << x << y;

cout <<“Position: “<< x << “, “ << y  << endl;

char command = ‘G’;
cout << command;
```

Assignment statement

Used to store results of computation into a variable. Form:

variable_name = expression;

•Example:

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

•Expression : can specify formula involving constants or variables, almost as in mathematics.

- If variables are specified, their values are used.
- multiplication 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.

Examples

```
int x=2, y=3, p=4, q=5, r, s, t;
x= r*s;  //disaster
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

```
int x=2, y=3, z, w;
float q=3., 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?
```

Evaluating "varA op varB"
e.g. x*q

- if varA, varB have same data type: result will have same data type.
- if varA, varB have different data types: result will have "more expressive" data type.
- int/short/unsigned int are less expressive than float/double
- shorter types are less expressive than longer.

Example

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

x/y : both are int. So truncation. Hence 0.
p/q : similarly 0.
p/y : 4/3 after truncation will be 1.
So prints 1.

Another example

```
int nsides=100,i_angle1,i_angle2;
i_angle1 = 360/nsides;
i_angle2 = 360.0/nsides;

float f_angle1, f_angle_2;
f_angle1 = 360/nsides;
f_angle2 = 360.0/nsides;
```

3

3

3.0

3.6

Example
(Impact of limited precision)

- float w,y=1.5, avogadro=6.022e23;
- w = y + avogadro;
- $6.022000 \times 10^{23} + 1.5$
- "Actual sum" : 602200000000000000000001.5
- 6.022000000000000000000015e23
- Sum will have to be stored in variable w of type float. But w can only accommodate 23 bits for significand, or about 7 digits.
  - ($2^{23} = 8388608$)
- 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.
}
```

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

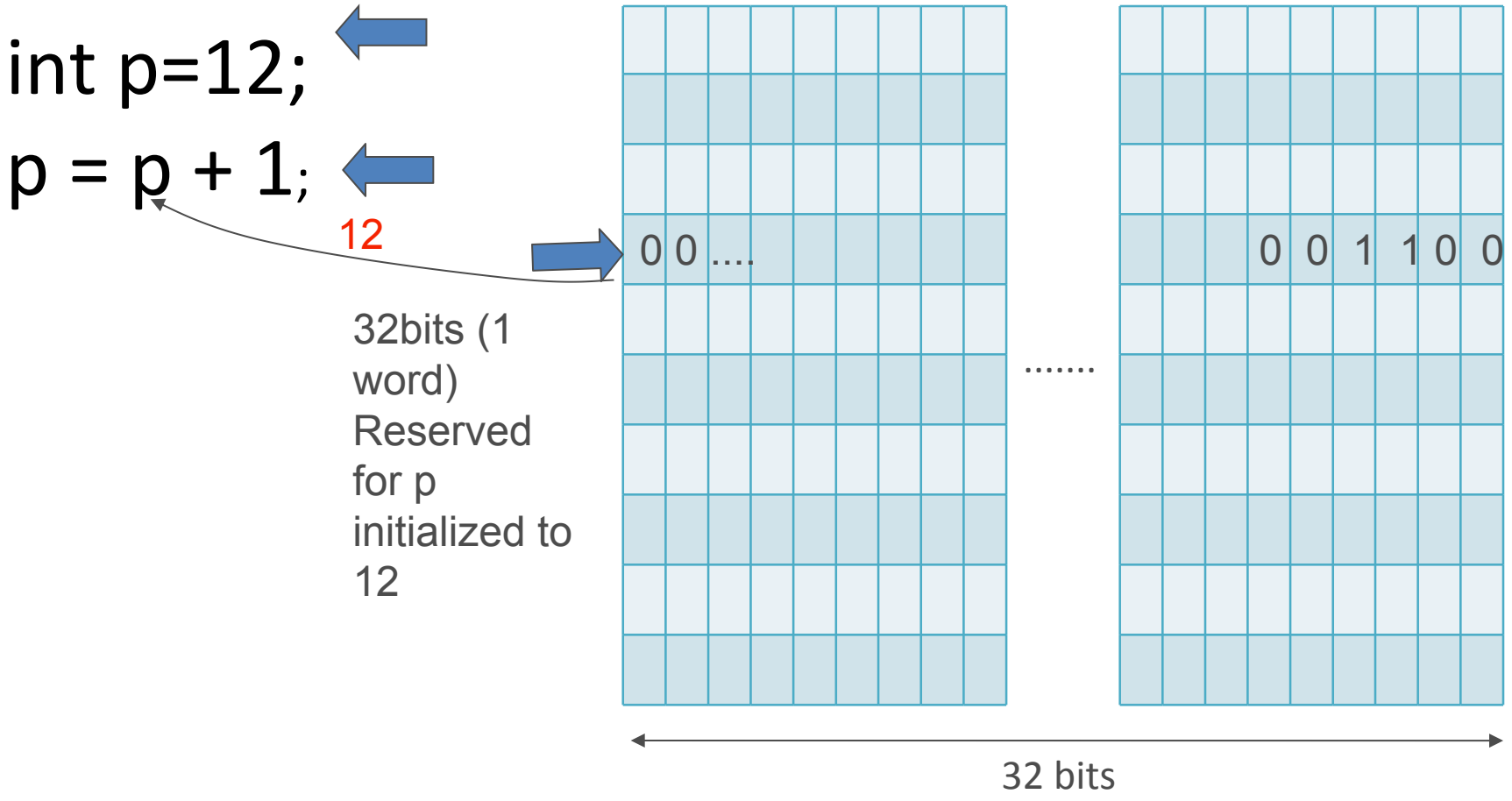
```
int p=12;
p = p + 1;
```

- Rule for evaluation: first evaluate the value on the right 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.

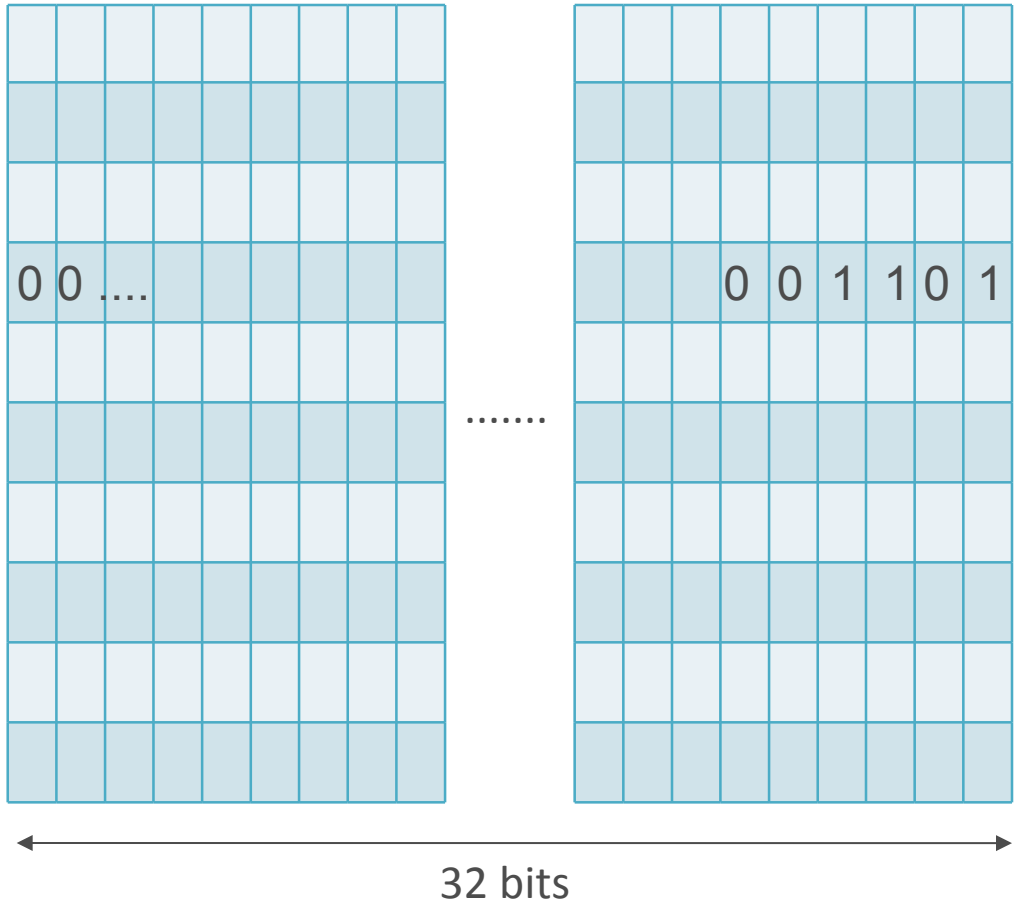
"=" in C++ is different from "=" in math.

An interesting assignment expression



An interesting assignment expression

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



The sum
12+1
is calculated
and stored
again in
p
itself
OLD VALUE
OVERWRITTEN

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

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, …?
- Both can be done by making slight modifications to previous program.

Another idiom: accumulation

```
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 other operators too.
```

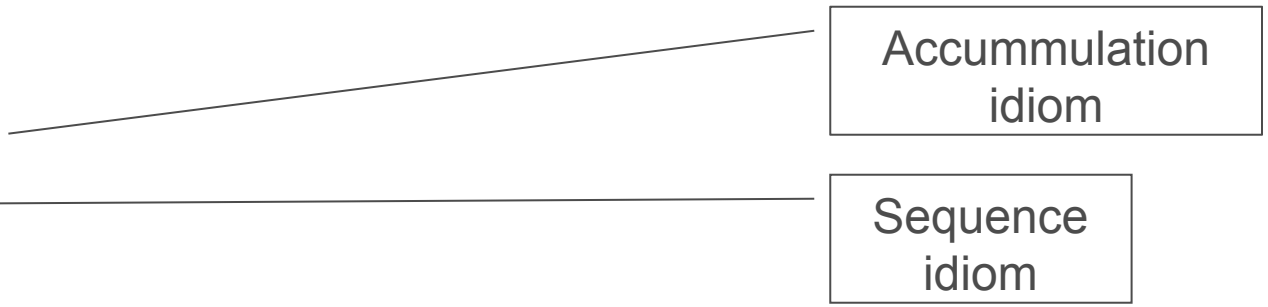
Composing the two idioms

Write a program to calculate n! given n.

Composing the two idioms

Write a program to calculate n! given n.

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



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.

int i=5, j=6;

++i; --j; // i becomes 6, j 5.

- ++ and -– can be put inside expressions, which is discussed in the book for completeness, but not recommended in polite programming.

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;
```

8

7

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

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 *=, -=, …
- 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
```

Blocks and Scope

- Code inside {} is called a block.
- Blocks are associated with repeats, but you may create them arbitrarily.
- 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!).

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;
    cout << x << endl; // prints 10
  }
   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).

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.