C++ Overview

- Designed by B. Stroustrup (1986)
- C++ and ANSI C (revised version of K&R C) are closely related
- Hybrid language: OO and ‘conventional’ programming
- More than just an OO version of C

Simple C++ Program

```
/* Example1: Compute the squares of both the sum and the
   difference of two given integers
*/
#include <iostream.h>
int main()
{
   cout << “Enter two integers: “; // Display
   int a, b;                       // request
   cin >> a >> b;           // Reads a and b
   int sum = a + b, diff = a - b,
       u = sum * sum, v = diff * diff;
   cout << “Square of sum        : “ << u << endl;
  cout << “Square of difference: “ << v << endl;
  return 0;
}
```

Key Points

- `/*  */`
  - begin and end of a comment
- `//`
  - beginning of a comment (ended by end of line)
- **`#include <iostream.h>`**
  - Includes the file iostream.h, a header file for stream input and output, e.g. the **<<** and **>>** operators
  - To include means to replace the include statement with the contents of the file
  - must be on a line of its own

Key Points

- In general, statements can be split over several lines
- Every C++ program contains one or more functions, one of which is called **`main`**

```
int main()      // no parameters here
  {       // beginning of body
    ...
  }       // end of body
```

- A function comprises statements which are terminated with a semi-colon

Key Points

- Declaration
  - Unlike C, a declaration is a normal statement and can occur anywhere in the function

```
int     sum = a + b, diff = a - b,
        u = sum * sum, v = diff * diff;
```

  - Declarations define variables and give them a type
  - Optionally, declarations initialize variables

Key Points

- Output to the ‘standard output stream’
  <<
- Input from the ‘standard input stream’
  >>
- Output of the end of a line is effected using the `endl` keyword
- Could also have used ‘`\n`’ or “`\n`”

Identifiers

- Sequence of characters in which only letters, digits, and underscore _ may occur
- Case sensitive ... upper and lower case letters are different

Identifiers

- Reserved identifiers (keywords):
  - `asm, auto, break, case, catch, char, class, const, continue, default, delete, do, double, else, enum, extern, float, for, friend, goto, if, inline, int, long, new operator, private, protected, public, register, return, short, switch, template, this, throw, try, typedef, union, unsigned, virtual, void, volatile, while`

Constants

- Integer constants
  - 123 (decimal)
  - 0777 (octal)
  - 0xFF3A (hexadecimal)
  - 123L (decimal, long)
  - 12U (decimal, unsigned)

Constants

- Character constants
  - ‘**A**’ enclosed in single quotes
  - Special characters (escape sequences)
    - ‘**\n**’ newline, go to the beginning of the next line
    - ‘**\r**’ carriage return, back to the beginning the current line
    - ‘**\t**’ horizontal tab
    - ‘**\v**’ vertical tab
    - ‘**\b**’ backspace
    - ‘**\f**’ form feed
    - ‘**\a**’ audible alert

Constants

- Character constants

'****'	backslash
'**\'**'	single quote
'**\"**'	double quote
'**\?**'	question mark
'***000***'	octal number
'**\x*hh***'	hex number

Constants

- Floating Constants
  - Type **`double`**

    82.247

    .63

    83.

    47e-4

    1.25E7

    61.e+4
  - Type **`float`**

    82.247L

    .63l

Constants

- Floating Constants

Type	Number of Bytes
`float`	4
`double`	8
`long double`	10

- Implementation dependent

Constants

- String Constants
  - String literal
  - String

    `“How many numbers?”`

    `“a”`
  - `“a”` is not the same as `‘a’`
  - A string is an array of characters terminated by the escape sequence `‘\0’`
  - Other escape sequences can be used in string literals, e.g. `“How many\nnumbers?”`

Constants

- String Constants
  - Concatenation of string constants

```
"How many numbers?"
```

is equivalent to

```
"How many"
    " numbers?"
```

  - This is new to C++ and ANSI C

Constants

- String Constants

```
cout << “This is a string that is \
regarded as being on one line”;
```

is equivalent to

```
cout << “This is a string that is”
        “regarded as being on one line”;
```

Comments

- `/* text of comment */`
- `//  text of comment`
- Within a comment, the characters sequences `/*. */,` and `//` have no meaning

  So comments cannot be nested
- Use

```
#if 0
code fragment to be commented out
...
#endif
```

Exercises

1. Write a program that prints your name and address. Compile and run this program
2. Write a program that prints what will be your age at the end of the year. The program should request you to enter both the current year and the year of your birth
3. Modify the program to print also your age at the end of the millenium

Exercises

4. Use the operator **<<** only once to print the following three lines:

```
One double quote: “
Two double quotes: ““
Backslash: \
```

Exercises

5. Correct the errors in the following program

```
include <iostream.h>
int main();
{
   int i, j
   i = ‘A’;
   j = “B”;
   i = ‘C’ + 1;
   cout >> “End of program”;
   return 0
}
```

Expressions and Statements

- Expressions

```
a + b
x = p + q * r
```

- Statements

```
a + b;
x = p + q * r;
```

- Operators

```
+, *, =
```

- Operands

```
a, b, p, q, r, x
```

Arithmetic Operations

- Unary operator: -, +

```
neg = -epsilon;
pos = +epsilon;
```

Arithmetic Operations

- Binary operators: `+, -, *, /, %`

```
a = b + c;
```

  - Integer overflow is not detected
  - Results of division depends on the types of the operands

```
float fa = 1.0, fb = 3.0;
int a = 1, b = 3;
cout << fa/fb;
cout << a/b;
```

Arithmetic Operations

- Remainder on integer division

  %

```
39 % 5    // value of this expression?
```

Arithmetic Operations

- Assignment and addition

```
x = x + a
x += a
```

  - These are expressions and yield a value as well as performing an assignment

```
y = 3 * (x += a) + 2;  //!!!
```

Arithmetic Operations

- Other assignment operators

```
x -= a
x *= a
x /= a
x %= a

++i     // increment operator: i += 1
--i     // decrement operator: i -= 1
```

Arithmetic Operations

- Other assignment operators

```
/* value of expression = new value of i */

++i     // increment operator: i += 1
--i     // decrement operator: i -= 1

/* value of expression = old value of i */

i++     // increment operator: i += 1
i--     // decrement operator: i -= 1
```

Types, Variables, and Assignments

Type	Number of Bytes
`char`	1
`short (short int)`	2
`int`	2
`enum`	2
`long (long int)`	4
`float`	4
`double`	8
`long double`	10

Types, Variables, and Assignments

- Use **`sizeof`** to find the size of a type

  e.g.

```
cout << sizeof (double)
```

Types, Variables, and Assignments

- **<<** doesn't allow user-specified formatting of output; use (C library function) `printf`

```
char ch = 'A'; int i = 0;
float f = 1.1; double ff = 3.14159;

printf("ch = %c, i = %d\n", ch, i);
printf("f = %10f, ff = %20.15f\n", f, ff);
```

Types, Variables, and Assignments

- To use **`printf`** you must include `stdio.h`

```
#include <stdio.h>
```

- syntax:

```
printf(<format string>, <list of variables>);
```

- `<format string>`
  String containing text to be printed and conversion specifications

Types, Variables, and Assignments

- Conversion specifications

```
%c          characters
%d          decimals
%f          floats or doubles
%s          strings
```

- can also include field width specifications:

```
%m.kf       m is the field width
            k is the number of digits
            after the decimal point
```

Types, Variables, and Assignments

- **>>** doesn't allow user-specification of input types; use (C library function) `scanf`

```
char ch = 'A'; int i = 0;
float f = 1.1; double ff = 3.14159;

scanf("%c %d %f %lf", &ch, &i, &f, &ff);
```

- The ampersand `&` is essential
  - It takes the address of the variable that follows
  - scanf expects only variables

Types, Variables, and Assignments

- Enumerated types enum
  - Used to define constant values whose names mean something but whose actual values are irrelevant

```
enum days
{ Sunday, Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday
} yesterday, today, tomorrow;
days the_day_after_tomorrow;
```

  - Sunday, ..., Saturday are symbolic integer constants, have values 0, .., 6, respectively and are the values of type days

scanf("%c %d %f %lf", &ch, &i, &f, &ff);

Types, Variables, and Assignments

- Enumerated types example

```
today = Monday;
the_day_after_tomorrow = Tuesday;
```

- C++ has no built-in logical or Boolean type
  - We can define one using enumerated types

```
enum Boolean {FALSE, TRUE};
```

Types, Variables, and Assignments

- Register variables
  - access to data in registers is generally faster than access to data in memory
  - We can ask to compiler to put very frequently used variables in a register:

```
register int i;
```

  - Cannot take the address of a register variable

```
scanf("%d", &i); // illegal operation
```

Types, Variables, and Assignments

- Use the type qualifier `const` to define constants

```
const int weeklength = 7;
```

  - The initialization of weeklength is essential since we cannot assign values to constants subsequently

```
weeklength = 7; // Error
```

Comparison and Logical Operators

Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
&&	logical AND
||	logical OR
!	logical NOT

Comparison and Logical Operators

- `<`, `>`, `<=`, `>=` are relational operators
- `==` and `!=` are equality operators
- relational operators have a higher precedence than equality operators
- Expression formed with these operators yield one of two possible values
  - 0 means false
  - 1 means true
  - Both are of type int

Compound Statement

- Statements describe actions
- Expressions yield values
- We use braces {} to build complex - compound - statement from simpler ones
- Typically, we use compound statements in places where the syntax allows only one statement

```
{x = a + b; y = a - b;}
```

Compound Statement

- Compound statements are called blocks
- A declaration in a block is valid from the point of declaration until the closing brace of the block
- The portion of the program corresponding to this validity is called the scope of the variable which has been declared
- Variables are only visible in their scope

Compound Statement

```
// SCOPE:  Illustration of scope and visibility
#include <iostream.h>
int main()
{ float x = 3.4;
  { cout << “x = “ << x << endl;
    // output: x = 3.4 (because float x is visible
    int x = 7;
    cout << “x = “ << x << endl;
    // output x = 7 (because int x is visible
    // float x is still in scope but hidden
    char x = ‘A’;
    cout << “x = “ << x << endl;
    // output x = A (because char x is visible
    // float x and int x are still in scope but hidden
  } // end of block
```

Compound Statement

```
    cout << “x = “ << x << endl;
    // output x = 3.4 (because char x is visible
    // int x and char x are out of scope
    return 0;
} // end of main
```

Conditional Statements

- Syntax

```
if (expression)
    statement1
else
    statement2
```

  - The else clause is optional
- Semantics
  - `statement1` is executed if the value of expression is non-zero
  - `statement2` is executed if the value of expression is zero

Conditional Statements

- Where appropriate *statement1* and *statement2* can be compound statements

```
if (a >= b)
{  x = 0;
   if (a >= b+1)
   {  xx = 0;
      yy = -1;
   }
   else
   {  xx = 100;
      yy = 200;
   }
 }
```

Iteration Statements

- while-statement syntax

```
while (expression)
    statement
```

- semantics
  - `statement` is executed (repeatedly) as long as `expression` is non-zero (true)
  - `expression` is evaluated before entry to the loop

Iteration Statements

```
// compute s = 1 + 2 + ... + n

s = 0;
i = 1;
while (i <= n)
{  s += i;
   i++;
}
```

Iteration Statements

- do-statement syntax

```
do
   statement
while (expression);
```

- semantics
  - `statement` is executed (repeatedly) as long as `expression` is non-zero (true)
  - `expression` is evaluated after entry to the loop

Iteration Statements

```
// compute s = 1 + 2 + ... + n

s = 0;
i = 1;
do                      // incorrect if n == 0
{  s += i;
   i++;
} while (i <= n)
```

Iteration Statements

- for-statement

  `for (`*`statement1`* *`expression2`*`; `*`expression3`*`)`
  `statement2`

- semantics
  - *`statement1`* is executed
  - *`statement2`* is executed (repeatedly) as long as *`expression2`* is true (non-zero)
  - *`expression3`* is executed after each iteration (i.e. after each execution of *`statement2`*)
  - *`expression2`* is evaluated before entry to the loop

Iteration Statements

```
// compute s = 1 + 2 + ... + n

s = 0;
for (i = 1; i <= n; i++)
   s += i;
```

Iteration Statements

```
for (statement1  expression2; expression3)
      statement2
```

- We have *statement1* rather than *expression1* as it allows us to use an initialized declaration

  `int i=0;`

  – Note that the for statement does not cause the beginning of a new block (and scope) so we can only declare a variable which has not already been declared in that scope.

  – The scope of the declaration ends at the next `}`

Iteration Statements

```
// compute s = 1 + 2 + ... + n

s = 0;
for (int i = 1; i <= n; i++)
   s += i;
```

Break and Continue

- `break;`
  - the execution of a loop terminates immediately if, in its inner part, the `break;` statement is executed.

Break and Continue

```
// example of the break statement

for (int i = 1; i <= n; i++)
{  s += i;
   if (s > max_int)  // terminate loop if
      break;         // maximum sum reached
}


/* Note:  there is a much better way */
/* to write this code                */
```

Break and Continue

- `continue;`
  - the continue statement causes an immediate jump to the text for continuation of the (smallest enclosing) loop.

Break and Continue

```
// example of the continue statement

for (int i = 1; i <= n; i++)
{  s += i;
   if ((i % 10) != 0)  // print sum every
      continue;        // tenth iteration
   cout << s;
}

/* Note:  there is a much better way */
/* to write this code                */
```

Switch

- `switch (expression) statement`
  - the `switch` statement causes an immediate jump to the statement whose label matches the value of `expression`
  - `statement` is normally a compound statement with several statements and several labels
  - `expression` must be of type `int`, `char`, or `enum`

Switch

```
// example of the switch statement

switch (letter)
{  case ‘N’: cout < “New York\n”;
             break;
   case ‘L’: cout < “London\n”;
             break;
   case ‘A’: cout < “Amsterdam\n”;
             break;
   default:  cout < “Somewhere else\n”;
             break;
}
```

Switch

```
// example of the switch statement

switch (letter)
{  case ‘N’: case ‘n’: cout < “New York\n”;
                       break;
   case ‘L’: case ‘l’: cout < “London\n”;
                       break;
   case ‘A’: case ‘a’: cout < “Amsterdam\n”;
                       break;
   default:  cout < “Somewhere else\n”;
             break;
}
```

Exercises

6. Write a program that reads 20 integers and counts how often a larger integer is immediately followed by a smaller one

Conditional Expressions

- conditional expression syntax

```
expression1 ? expression2 : expression3
```

- semantics
  - if the value of *expression1* is true (non-zero)
  - then *expression2* is evaluated and this is the value of the entire conditional expression
  - otherwise *expression3* is evaluated and this is the value of the entire conditional expression

conditional expression

```
// example of the conditional expression

z = 3 * (a < b ? a + 1 : b - 1) + 2;

// alternative

if (a < b)
   z = 3 * (a + 1) + 2;
else
   z = 3 * (b - 1) + 2;
```

conditional expression

```
// example of the conditional expression

cout << “The greater of a and b is” <<
                                (a > b ? a : b);

// alternative
cout << “The greater of a and b is”
if (a < b)
   cout << a;
else
   cout << b;
```

The Comma-operator

- comma-operator syntax

  expression1 , *expression2*

- semantics
  - *expression1* and *expression2* are evaluated in turn and the value of the entire (compound) expression is equal to the value of *expression2*

The Comma-operator

```
// example of the comma operator
// compute sum of input numbers

s = 0;
while (cin >> i, i > 0)
   s += i;

// or ...
s = 0;
while (scanf (“%d”, &i), i > 0)
   s += i;
```

The Comma-operator

```
// Note that here scanf() is used as an
// expression and yields a value ... the
// number of successfully-read arguments

s = 0;
while (scanf (“%d”, &i) == 1) // terminate on
    s += i;                   // non-integer
                              // input
```

Bit Manipulation

- The following bit manipulation operators can be applied to integer operands:

&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Inversion of all bits
<<	Shift left
>>	Shift right

- Note, in C++, the meaning of an operator depends on the nature of its operands (cf &, <<, >>)

Simple Arrays

- The array declaration

  ```
  int a[100]
  ```

  enables us to use the following variables:

  ```
  a[0], a[1], ... a[99]
  ```

  each element being of type `int`

Simple Arrays

- subscripts can be an integer expression with value less than the array size (e.g. 100)
- In the declaration, the dimension must be a constant expression

```
#define LENGTH 100
...
int a[LENGTH]
...
for (int i=0; i<LENGTH; i++)
   a[i] = 0;  // initialize array
```

Simple Arrays

- Alternatively

```
const int LENGTH = 100;
...
int a[LENGTH]
...
for (int i=0; i<LENGTH; i++)
   a[i] = 0;  // initialize array
```

Simple Arrays

```
// LIFO: This program reads 30 integers and
// prints them out in reverse order: Last In, First Out
#include <iostream.h>
#include <iomanip.h>

int main()
{ const int LENGTH = 30;
 int i, a[LENGTH];
 cout << “Enter “ << LENGTH << “ integers:\n”;
for (i=0; i<LENGTH; i++) cin >> a[i];
cout << “\nThe same integers in reverse order:\n”;
for (i=0; i<LENGTH; i++)
   cout << setw(6) << a[LENGTH - i - 1]
        << (i % 10 == 9 ? ‘\n’ : ‘ ‘);
return 0;
}
```

Simple Arrays

- Initializing an array

```
const int LENGTH = 4;
...
int a[LENGTH] = {34, 22, 11, 10};

int b[LENGTH] = {34, 22}; // element 2, 3 = 0

int c[20] = “Tim”;
            // same as = {‘T’, ‘i’, ‘m’, ‘\0’ }
```

Associativity

- Most operators are left-associative

```
a - b * c      //    ((a - b) * c)
               // or (a - (b * c))
```

- Right-associative operators
  - all unary operators
  - the operator `?:`, used in expressions
  - the assignment operators
    `=, +=, *=, /=, %=, &=, |=, ^=, <<=, >>=`

Associativity

- example

```
-n++    //  value for n=1? -2 or 0
```

Precedence of Operators

- operators in order of decreasing precedence (same precedence for same line)

```
() [] . -> ::
! ~ ++ + - (type) * &  sizeof new delete // all unary
.*   ->*
*   /   %
+   -
<<   >>
<   <=   >   >=
==   !=
```

Precedence of Operators

```
&
^
|
&&
||
?:
=   +=   -=   *=   /=   %=   &=   |=   ^=   <<=   >>=
,
```

Precedence of Operators

Operator	Meaning
::	scope resolution
()	function calls
[]	subscripting
.	selecting a component of a structure
->	selecting a component of a structure by means of a pointer
.*	pointers to class members
->*	pointers to class members
!	NOT, unary operator
~	inversion of all bits, unary operator

Precedence of Operators

Operator	Meaning
++	increment, unary operator
--	decrement, unary operator
+	plus, unary operator
+	addition, binary operator
-	minus, unary operator
-	minus, binary operator
(type)	cast, unary operator
new	create (allocate memory)
delete	delete (free memory)
*	‘contents of address’, unary operator
*	multiplication, binary operator

Precedence of Operators

Operator	Meaning
&	bitwise AND, binary operator
&	‘address of’, unary operator
sizeof	number of bytes inm memory, unary operator
/	division, either floating point or integer
%	remainder with integer division
<<	shift left; stream output
>>	shift right; stream input
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Precedence of Operators

Operator	Meaning
==	equal to
!=	not equal to
^	bitwise exclusive OR (XOR)
|	bitwise OR
&&	logical AND
||	logical OR
?:	conditional expression
=	assignment
+=	addition combined with assignent (other operators can also be combined with assignment)

Arithmetic Conversions

- Every arithmetic expression has a type
- This type can be derived from those of its operands
  - first, integral promotion may take place: operands of type `char`, `short`, and `enum` are ‘promoted’ to `int` if this type can represent all the values of the orginal type; otherwise the original type is converted to `unsigned int`
  - type conversion is now applied, as follows

Arithmetic Conversions

- One of the following 7 rules is applied (considering each in strict order)
  - If either operand is `long double`, the other is converted to this type
  - If either operand is `double`, the other is converted to this type
  - If either operand is `float`, the other is converted to this type
  - If either operand is `unsigned long`, the other is converted to this type

Arithmetic Conversions

- One of the following 7 rules is applied (considering each in strict order)
  - If either operand is `long` and the other is `unsigned`, the other is converted to `long`, provided that `long` can represent all the values of `unsigned`. If not, both operands are converted to `unsigned long`
  - If either operand is a `long`, the other is converted to this type
  - If either operand is a `unsigned`, the other is converted to this type

The cast-operator

- Forced type conversion
  - casting
  - coercion
- `(float)n // cast n as a float (C and C++)`
- `float(n) // cast n as a float (C++)`
- Example

```
int i=14, j=3;
float x, y;
x = i/j;                    // x = 4.0
y = float(i)/float(j);      // y = 4.666..
y = float(i/j);             // y = 4.0
```

The cast-operator

- Example

```
int i;
float x = -6.9;
i = x;          // i = -6
i = int(x);     // i = -6, but it is clear
                // that conversion takes
                // place
```

Lvalues

- Consider an assignment expression of the form:

```
E1 = E2
```

- Normally `E1` will be a variable, but it can also be an expression
- An expression that can occur on the left-hand side of an assigment operator is called a *modifiable lvalue*

Lvalues

- Not lvalues:

```
3 * 5
i + 1
printf(“&d”, a)
```

- lvalues (given `int i, j, a[100], b[100];`)

```
i
a[3 * i + j]
(i)
```

Lvalues

- Array names are not lvalues:

```
a = b; // error
```

- lvalues

```
(i < j ? i : j) = 0;// assign 0 to
                    // smaller of i and j
```

  - Since `?:` has higher precedence than =, we can write as:

```
i < j ? i : j = 0;  // !!!
```

Lvalues

- The conditional expression `E1 ? E2 : E3` is an lvalue only if `E2` and `E3` are of the same type and are both lvalues
  - NB: this is a C++ feature; conditional expressions cannot be lvalues in C
- The results of a cast is not an lvalue (except in the case of reference types, yet to come)

```
float(x) = 3.14; // error
```