Intro to C++ Language

Scalar Variables, Operators

and

Control Structures

Structure of a C++ Program

- A C++ program is a collection of definitions and declarations:
  - data type definitions
  - global data declarations
  - function definitions (subroutines)
  - class definitions
  - a special function called **`main()`** (where the action starts).

Procedural vs. Object Oriented

- Procedural languages express programs as a collection of procedures (subroutines).
- Object Oriented languages express programs as a collection of object types (classes).

- C++ is both!
- We will start with *procedural* and build up to Object Oriented

Hello World++

```
// Hello World program

#include <iostream.h>

int main() {

  cout << "Hello World\n";

  return 0;
}
```

comment

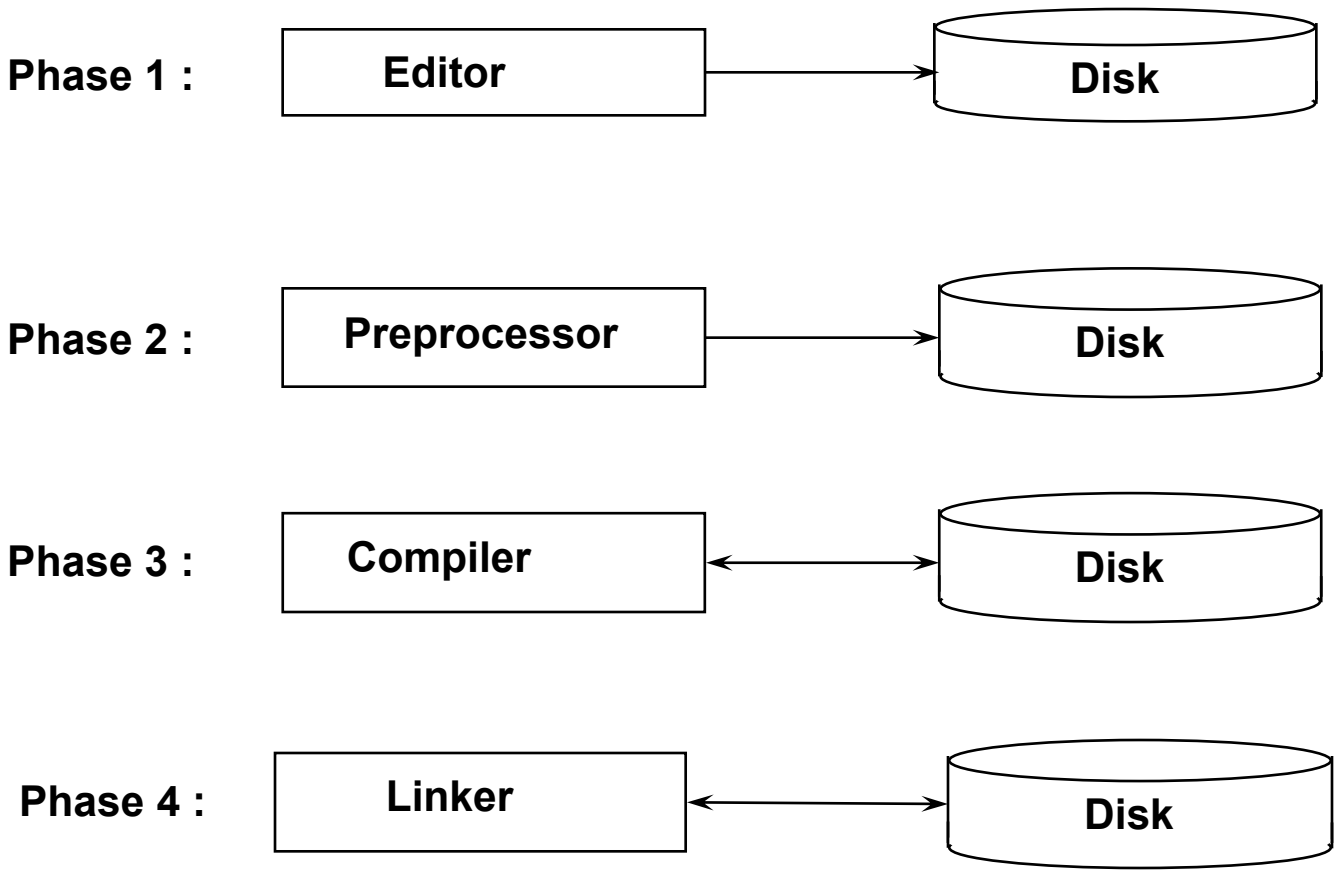
Allows access to an I/O library

Starts definition of special function main()

output (print) a string

Program returns a status code (0 means OK)

C++ Development Environment



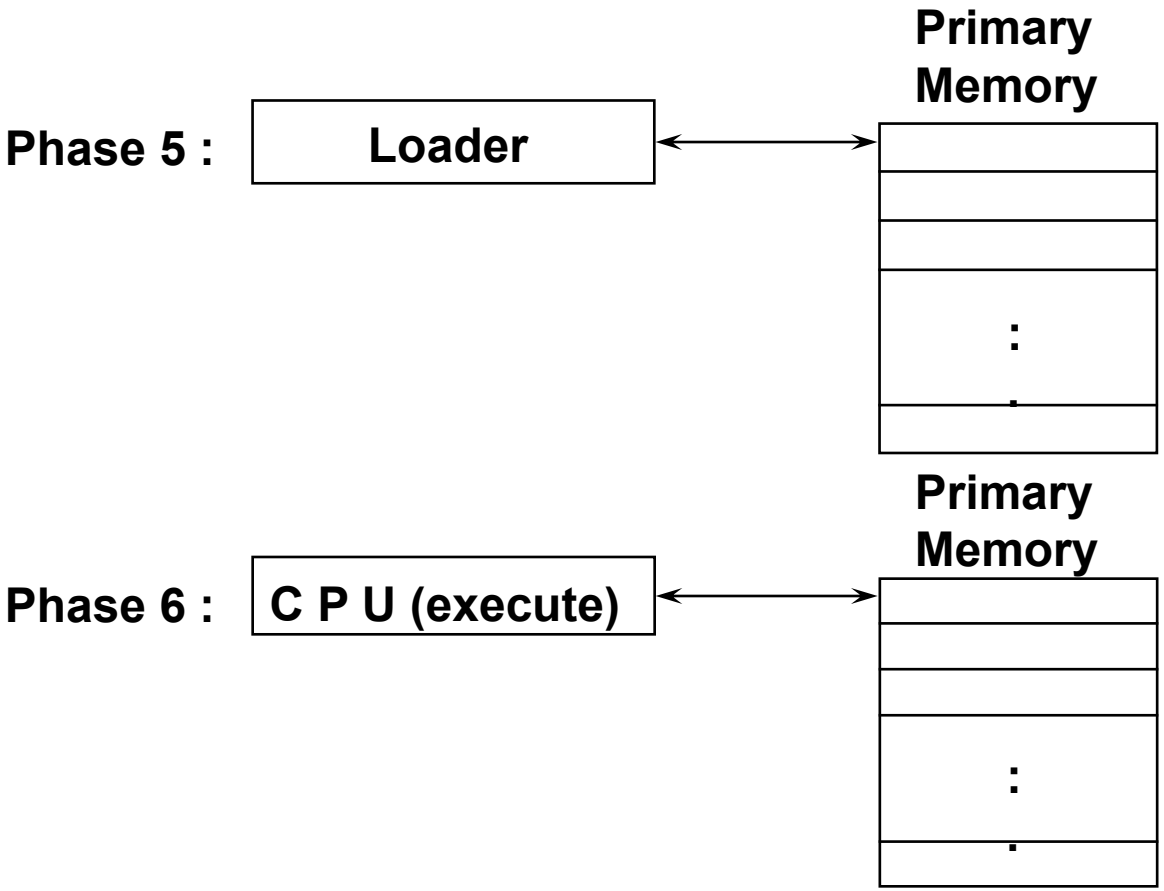
Program is created using the Editor and stored on Disk.

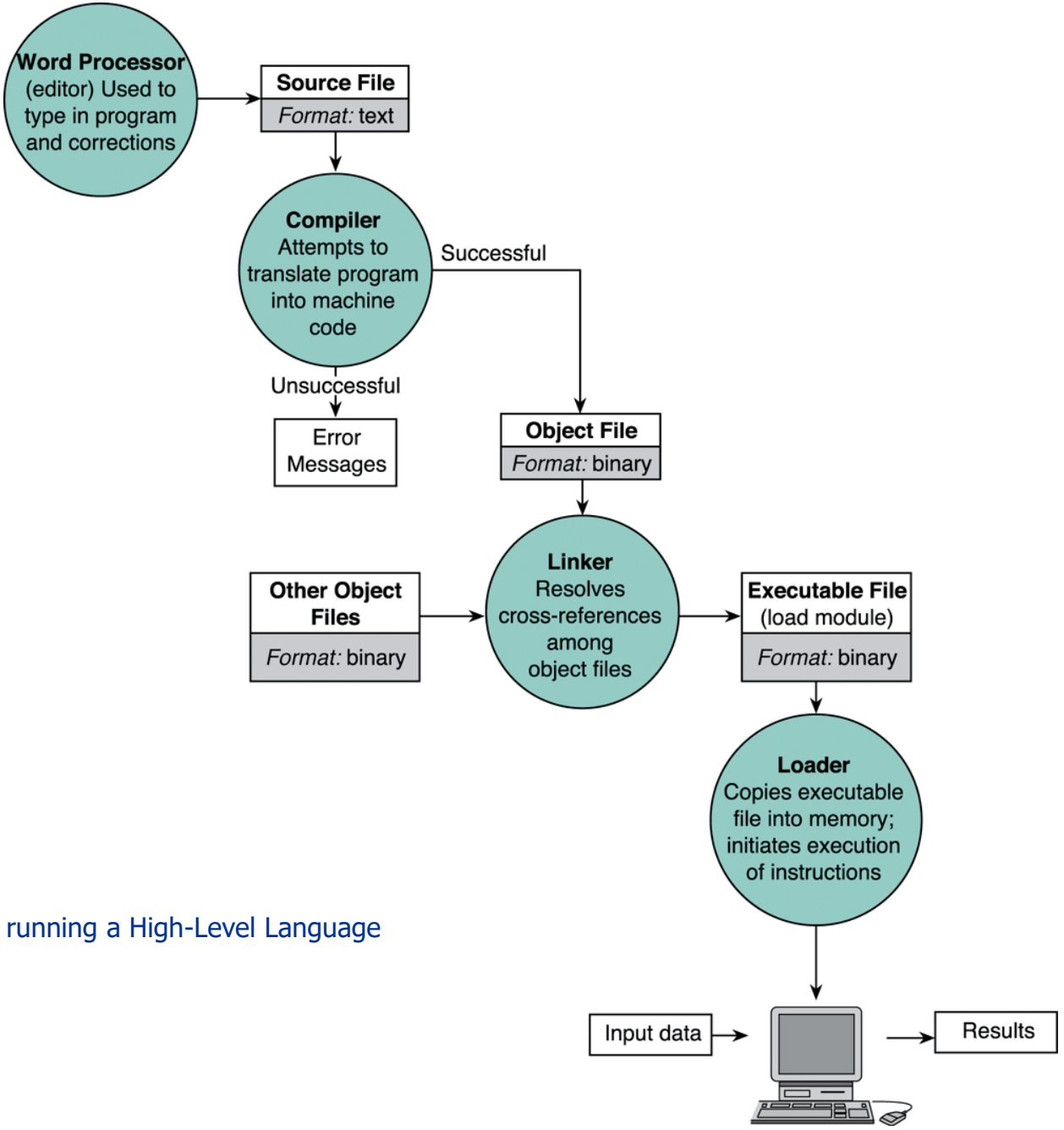
Preprocessor program processes the code.

Compiler creates object code and stores it on Disk.

Linker links object code with libraries, creates **a.out** and stores it on Disk

C++ Development Environment





Entering, translating, and running a High-Level Language Program

Comments

- Comments contain text that is not converted to machine language (it's just there for humans).
- Everything after "//" is ignored by the compiler.
- Everything between "/*" and "*/" is ignored.

Comment Example

```
// Dave's Homework #1
// This program is awesome!
#include <iostream.h>
/* This program computes the
   coefficient of expansion of the
   universe to 27 decimal places.
*/
int main() {
  cout << 1.000000000000000000000001;
}
```

Includes

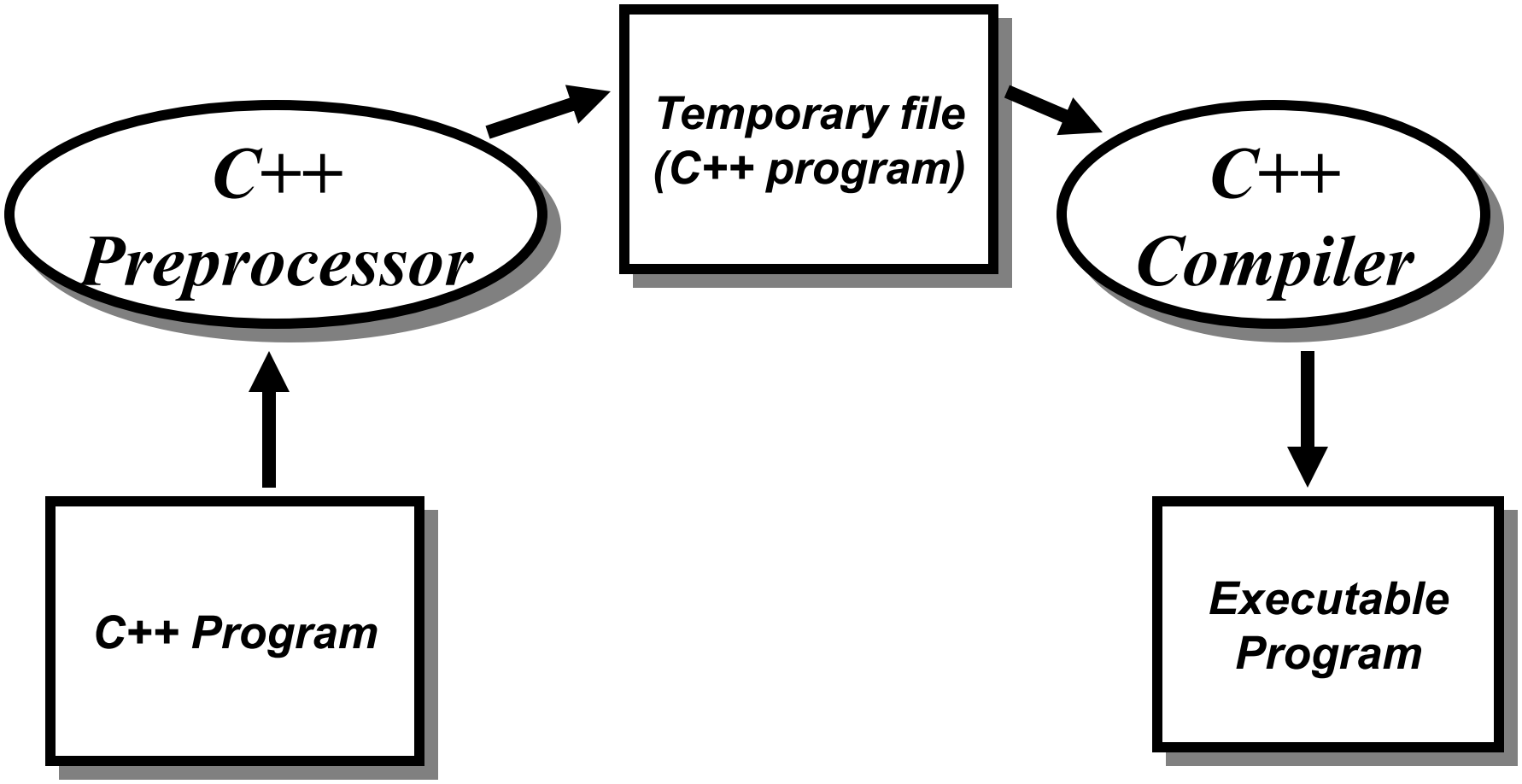
- The statement: `#include <foo.h>` inserts the contents of the file foo.h inside your file before the compiler starts.
- Definitions that allow your program to use the functions and classes that make up the standard C++ library are in these files.
- You can include your own file(s):

```
#include "myfile.h"
```

C++ Preprocessor

- C++ Compilers automatically invoke a *preprocessor* that takes care of #include statements and some other special directives.
- You don't need to do anything special to run the preprocessor - it happens automatically.

Preprocessing



The Preprocessor

- Lines that start with the character '# ' are special to instruct to a *preprocessor*.
- The preprocessor can replace the line with something else:
  - include: replaced with contents of a file
- Other *directives* tell the preprocessor to look for patterns in the program and do some fancy processing.

Preprocessor

1. File Inclusion (#include): This directive is used to insert the content of specified header files into the current source file. This allows for code reusability and access to declarations of functions, classes, and variables defined in other files.

- #include <iostream> *// Includes the iostream standard library header*
  #include "my_header.h" // Includes a user-defined header file

Some common includes

- Basic I/O: `iostream.h`
- I/O manipulation: `iomanip.h`
- Standard Library: `stdlib.h`
- Time and Date support: `time.h`

#define (macro) Example

#define allows you to define macros, which are essentially text substitutions. When the preprocessor encounters a defined macro, it replaces it with its corresponding replacement text.

```
#define square(a) (a * a)
y = square(x);
```

becomes `y = (x * x);`

```
z = square(y*x);
```

becomes `z = (y*x * y*x);`

Macro

- #define PI 3.14159 *// Defines a constant macro*
  #define MAX(a, b) ((a) > (b) ? (a) : (b))

Others

- #undef is used to remove a previously defined macro.
- Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, #endif): These directives allow you to conditionally compile parts of your code based on certain conditions. This is useful for:
- Including or excluding code based on build configurations (e.g., debug vs. release).
- Adapting code for different operating systems or platforms.
- Preventing multiple inclusions of header files (using #ifndef and #define with include guards).

Error Directives (#error, #warning):

- These directives are used to generate custom error or warning messages during the preprocessing phase, which can be helpful for debugging or enforcing coding standards.

Pragma Directives (#pragma):

- These directives provide a way to give specific instructions to the compiler, often related to compiler-specific features or optimizations. A common example is #pragma once for ensuring a header file is included only once.

Another C++ Program

```
// C++ Addition of integers
#include <iostream.h>

int main() {
   int integer1, integer2, sum;

   cout << "Enter first integer\n";
   cin >> integer1;
   cout << "Enter second integer\n";
   cin >> integer2;
   sum = integer1 + integer2;
   cout << "Sum is " << sum << endl;
   return 0;
}
```

Variables

- The program now uses *variables:*

```
int integer1, integer2, sum;
```

- Variables are just names for locations in memory.
- In C++ all variables must have a *type* (not all languages require this).
- In C++ all variables must be *declared* before they can be used.

Variables (cont.)

- C++ variables are declared like this:

```
type var_name;
```

- *type* indicates what kind of variable.
- C++ built in types include:

```
int char float double bool
```

- You can also create new types!

Variable Names

- C++ variable names:
  - made up of letters, digits and underscore.
  - Must start with a non-digit.
  - Case sensitive
    - `foo` is not the same name as `Foo`
- Can be any length
- *Good variable names tell the reader what the variable is used for!*

Literals (Constants)

- Literals are fixed values used by a program.
- Some examples of literals:

```
22      3.14159      0x2A
false      "Hi Dave"      'c'
```

- You can initialize a variable in the declaration by *assigning* it a value:

```
int foo = 17;
double PI = 3.14159;
char newline = '\n';
```

Expressions

- C++ *expressions* are used to express computation.
- Expressions include operations and the *operands* on which the operations are applied.
- Operands can be variables, literals or function calls.

Math Expressions

- Mathematical expressions have numeric values when evaluated.
- Some examples:

```
1+2
(fahr - 32)*(5/9)
1*(2*(3*(4*5)))
```

Mathematical Operators

```
+    -    *    /    %
```

- Operators have rules of *precedence* and *associativity* that control how expressions are evaluated.
- What is the value of this C++ expression ?:

```
2 / 3 / 4 + 5
```

- Answer: You can't tell unless you know the rules.

Associativity

- The associativity of an operator control the order of evaluation of expressions involving the same operator, for example:

```
3 / 4 / 5
```

- Associativity can be:
  - left-to-right: the leftmost operator is applied first.
  - Right-to-left: the rightmost operator is applied first.

Precedence

- Precedence controls the order of evaluation of operators.
  - A high precedence means an operator is evaluated (applied) before any lower precedence operators.
- Operators that have the same precedence can happen in either order, but in C++ the one on the left is evaluated first.

Operators

- An operator is a symbol that tells the computer to perform certain manipulations.
- An expression is a sequence of operands and operators that reduces to a single value.
- C operators can be classified into a number of categories.
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Assignment operators
  - Increment and decrement operators
  - Conditional operators
  - Bitwise operators
  - Special operators

Arithmetic operators

- The arithmetic operators in C

Operator	meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	modulo division

Arithmetic operators

- Note:,
  - Integer division truncates remainder
  - The % operator cannot be applied to a float or double.
  - The precedence of arithmetic operators
    - Unary + or -
    - * / %
    - + -

Arithmetic expressions

- An arithmetic expression is a combination of variables, constants, and operators.
- For example,
- a*b-c → a*b-c
- (m+n)(x+y) → (m+n)*(x+y)
- ax^2+bx+c → a*x*x+b*x+c

Shortcut Assignment Operators

Operator	*Example*	*Equivalent*
`+=`	`i+=8`	`i = i+8`
`-=`	`f-=8.0`	`f = f-8.0`
`*=`	`i*=8`	`i = i*8`
`/=`	`i/=8`	`i = i/8`
`%=`	`i%=8`	`i = i%8`

Relational Operators

- The relational operators in C are :

Operator	Meaning
<	less that
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Relational Operators

- A relational expression yields a value of 1 or 0.
  - $5 < 6$ 1
  - $-34 + 8 > 23 - 5$ 0
  - if a=3, b=2, c =1; then $a > b > c$ is ?
- the associativity of relational operators is left → right

Bitwise operators

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive
<<	Shift left
>>	Shift right

Result of logical Bitwise Operation

op1	op2	op1&op2	op1|op2	op1^op2
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Example

Bitwise AND

x - - -> 0000 0000 0000 1101
y - - -> 0000 0000 0001 1001

x&y- - -> 0000 0000 0000 1001

Bitwise OR

x - - - > 0000 0000 0000 1101
y - - -> 0000 0000 0000 1001
x|y - - -> 0000 0000 0001 1101

Bitwise Exclusive OR

x - - -> 0000 0000 0000 1101

y - - -> 0000 0000 0001 1001

x^y - - -> 0000 0000 0001 0100

Bitwise shift operators

- Left shift

  op<<n

Eg. 0100 1001 1100 1011

x<<3 0100 1110 0000

- Right shift

  op>>n

Bitwise Complement Operators

- ~ one's complement operator is a unary operator.

Logical operators

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Examples

Logical AND (&&)

a > b && x = = 10

Logical OR (||)

a < m || a < n

Logical NOT (!)

! (x >= y)

Assignment Operators

In addition, C has a set of shorthand assignment operators of the form.

var oper = exp;

Example

x = a + b

Increment and Decrement

syntax:

1. ++variable name
2. variable name++
3. – –variable name
4. variable name– –

x= 5;
y = ++x; (prefix)

In this case the value of y and x would be 6

x= 5;
y = x++; (post fix)

Then the value of y will be 5 and that of x will be 6.

Special operators

- Comma operator
- Size of operator
- Pointer operators (& and *)
- Member selection operators (. and ->).

Comma operator

- Link related expressions together
- Expressions are evaluated left to right

for e.g.

```
value = (x = 10, y = 5, x + y);
for (n=1, m=10, n <=m; n++, m++)
```

The sizeof Operator

- Gives size of bytes occupied in the memory.
- To determine the lengths of arrays and structures when their sizes are not known to the programmer

  for e.g.

  x = sizeof (sum);
  y = sizeof (long int);
  z= sizeof (235L);

Precedence and Associativity

Precedence rules decides the order in which different operator are applied.

Associativity rule decides the order in which multiple occurrences of the same level operator are applied.

Summary of C operator

Description	Operator	Rank	Associativity
Function call Array element reference	( ) []	1	Left to right
Unary plus Unary minus Increment Decrement Logical negation Ones complement Address Size of an object	+ - ++ -- ! ~ & Sizeof	2	Right to left
Multiplication Division Modulus	* / %	3	Left to right
Addition Subtraction	+ -	4	Left to right
Left shift Right shift	<< >>	5	Left to right
Less than Less than equal to Greater than Greater than equal to	< <= > >=	6	Left to right
Equality Inequality	= = |=	7	Left to right

Continue….

Bitwise AND	&	8	Left to right
Bitwise XOR	^	9	Left to right
Bitwise OR	|	10	Left to right
Logical AND	&&	11	Left to right
Logical OR	||	12	Left to right
Conditional operator	?:	13	Right to left
Assignment operator	= *=/=%= +=-=&= ^=|= << = >>=	14	Right to left
Commas operator	,	15	Left to right

Another Program

```
#include <iostream.h>

int main()
{
  double fahr,celcius;

  cout << "Enter Temperature in Fahrenheit\n";
  cin >> fahr;

  celcius = (fahr - 32.0)*5/9;
  cout << fahr << " fahrenheit is " << celcius
       << " Celcius" << endl;

  return 0;
}
```

const

- You can add the const modifier to the declaration of a variable to tell the compiler that the value cannot be changed:

```
const double factor = 5.0/9.0;
const double offset = 32.0;
celcius = (fahr - offset)*factor;
```

What if you try to change a `const`?

- The compiler will complain if your code tries to modify a const variable:

```
const foo = 100;
…
foo = 21;
```

Error: l-value specifies const object

Why use `const`?

- Const tells the compiler that a variable should never be changed.
- You already know the variable should never be changed!

- But - let the compiler save you from yourself (you might forget that it shouldn't be changed).

Integer vs. floating point math

- How does C++ *know* whether to use floating point or integer math operators?
- If either operand is floating point, a floating point operation is done (the result is a floating point value).
- If both operand are integer the result is an integer (even division).

Conditional operator

- a ternary operator pair “? : ” is available in C to construct conditional expressions of the form

  expr1 ? expr2 : expr3

- the expression *expr1* is evaluated first. If it is non-zero (true), then the expression *expr2* is evaluated, and that is the value of the conditional expression. Otherwise *expr3* is evaluated, and that is the value. Only one of *expr2* and *expr3* is evaluated.

Special operators

- **1. The Comma Operator**
- The comma operator can be used to link the related expressions together. A comma-linked list of expressions is evaluated left to right and the value of right-most expression is the value of the combined expression. For example, the statement
- value = (x=10, y=5, x+y);
- first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary.

Math Operator Quiz

What are the values printed?

```
const int five = 5;
int i = 7;
float x = 7.0;
cout << five + i/2 << endl;
cout << five + x/2 << endl;
```

Control Structures

- Unless something special happens a program is executed sequentially.
- When we want something special to happen we need to use a control structure.
- Control Structures provide two basic functions: selection and repetition

Selection

- A Selection control structure is used to choose among alternative courses of action.
- There must be some *condition* that determines whether or not an action occurs.
- C++ has a number of selection control structures:

if if/else switch

Repetition Control Structures

- Repetition control structures allow us to repeat a sequence of actions (statements).

- C++ supports a number of repetition control structures:

  while for do/while

`if`

- The `if` control structure allows us to state that an action (sequence of statements) should happen only when some condition is true:

```
if (condition)
    action;
```

Conditions

- The *condition* used in an `if` (and other control structures) is a Boolean value - either `true` or `false`.
- In C++:
  - the value 0 is `false`
  - anything else is `true`

`if` examples

```
if (1)
   cout << "I am true!\n";

if (1-1)
   cout << "I am false!\n";
```

Relational and Equality Operators and Conditions

- Typically a condition is built using the C++ relational and equality operators.
- These operators have the values `true` (1) and `false` (0).
- So the expression `x==x` has the value `true`.
- and `7 <= 3` has the value false.

More **if**s

```
if (foo)
   cout << "foo is not zero\n";

if (grade>=90)
  lettergrade = 'A';

if (lettergrade == 'F')
  cout << "The system has failed you\n"
```

Common Mistake

- It is easy to mix up the assignment operator "=" with the equality operator "==".
- What's wrong with this:

```
if (grade=100)
   cout << "your grade is perfect -
     RPI has decided to give you your
     degree today!\n";
```

Compound Statements

- Inside an `if` you can put a single statement or a compound statement.
- A compound statement starts with "{", ends with "}" and can contain a sequence of statements (or control structures)

```
if (grade>=90) {
   cout << "Nice job - you get an A\n";
   acnt = acnt+1;
}
```

A word about style

- C++ doesn't care about whitespace (including newlines), so you can arrange your code in many ways.
- There are a couple of *often-used* styles.
- All that is important is that the code is easy to understand and change!

Some common styles

```
if (foo>10) {
  x=y+100;
  cout << x;
}
```

```
if (foo>10)
  {
  x=y+100;
  cout << x;
  }
```

```
if(foo>10){x=y+100;cout<<x;}
```

`if else` Control Structure

- The `if else` control structure allows you to specify an alternative action:

```
if ( condition )
     action if true
else
     action if false
```

`if else` example

```
if (grade >= 90)
  lettergrade = 'A';
else
  lettergrade = 'F';
```

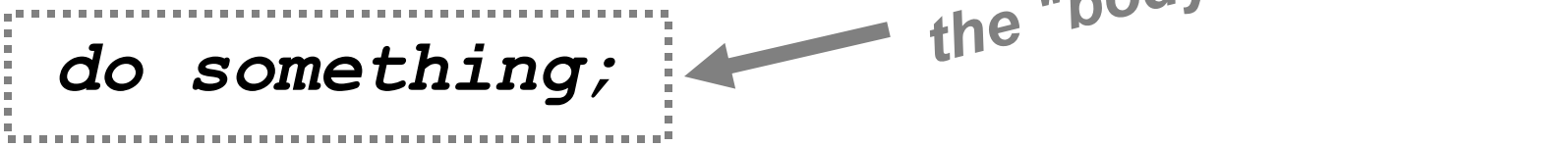
Another example

```
if (grade >= 99)
  lettergrade = 'A';
else if (grade >= 98)
  lettergrade = 'B';
else if (grade >= 97)
   lettergrade = 'C';
else if (grade >= 96)
   lettergrade = 'D';
else
   lettergrade = 'F';
```

`while` Control Structure

- The `while` control structure supports repetition - the same statement (or compound statement) is repeated until the condition is false.

```
while (condition)
  do something;
```



`while` dangers

- The body of the loop must change something that effects the *condition*!

- if not - the loop could never end
  - or the loop would never end
    - or the loop would never end
      - or the loop would never end
        - » or the loop would never end

          …….

`while` example

```
lettergrade = 'A';
cutoff = 90;
while (grade < cutoff) {
  lettergrade = lettergrade + 1;
  cutoff = cutoff - 10;
}
if (lettergrade > 'F')
  lettergrade = 'F';
```

Off topic - increment and decrement operators

- You can increment an integer variable like this:

```
// same as lettergrade = lettergrade + 1;
lettergrade++;
```

- You can also decrement:

```
// same as lettergrade = lettergrade - 1;
lettergrade--;
```

More off topic - special assignment operators

- This C++ statement:

```
foo += 17;
```

  – is shorthand for this:

```
foo = foo + 17;
```

- You can also use:

```
-=    *=    /=
```

`while` example modified

```
lettergrade = 'A';
cutoff = 90;
while (grade < cutoff) {
  lettergrade++;
  cutoff -= 10;
}
if (lettergrade > 'F')
  lettergrade = 'F';
```

do while

- The `do while` control structure also provides repetition, this time the condition is at the bottom of the loop.
  - the body is always executed at least once

```
do
   somestuff;
while ( condition );
```

do example

```
i=1;
do
  cout << "i is " << i++ << endl;
while (i <= 10);
```

`for` loops

- The `for` control structure is often used for loops that involve counting.
- You can write any `for` loop as a `while` (and any `while` as a `for`).

```
for (initialization; condition; update)
   dosomething;
```

```
for (initialization; condition; update)
```

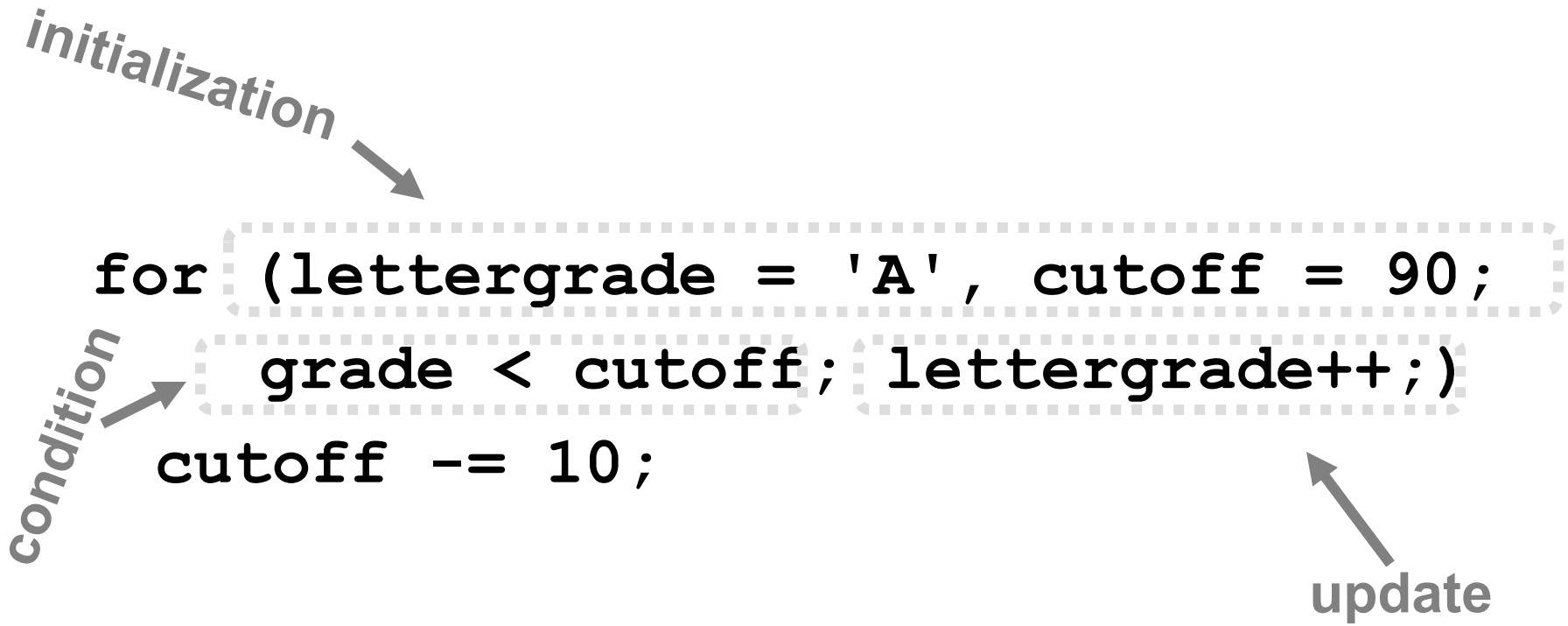
- initialization is a statement that is executed at the beginning of the loop (and never again).
- the body of the loop is executed as long as the condition is true.
- the update statement is executed each time the body of the loop has been executed (and before the condition is checked)

for example

```
for (i=1; i<10; i++)
  cout << "i is " << i << endl;

for (i=10; i>=0; i--)
  cout << "i is " << i << endl;
```

another `for`




```
for (lettergrade = 'A', cutoff = 90;
     grade < cutoff; lettergrade++;)
  cutoff -= 10;

if (lettergrade > 'F')
  lettergrade = 'F';
```

Yet another odd example

```
for (i=1; i<100;i++) {
  cout << "Checking " << i << endl;
  if ( i%2 )
     cout << i << " is odd" << endl;
  else
     cout << i << " is even" << endl;
}
```

More about `for`

- You can leave the initialization, condition or update statements blank.
- If the condition is blank the loop never ends!

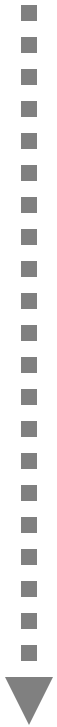
```
for (i=0; ;i++)
   cout << i << endl;
```

Complex Conditions

- You can build complex conditions that involve more than one relational or equality operator.
- The `&&` (means "and") and `||` (means "or") operators are used to create complex conditions.
- More operators means another precedence table...

Updated Precedence Table

Operators	Precedence
`()`	highest (applied first)
`++ --`	
`* / %`	
`+ -`	
`< <= > >=`	
`== !=`	
`&&`	
`||`	
`=`	lowest (applied last)



`&&` Operator

- `&&` is a boolean operator, so the value of an expression is `true` or `false`.

(*cond1* `&&` *cond2*)

is true only if both *cond1* and *cond2* are true.

&& Example

```
lettergrade = 'A';
cutoff = 90;
while ((grade<cutoff) && (lettergrade!='F')) {
  lettergrade++;
  cutoff -= 10;
}
```

|| Operator

- `||` is a boolean operator, so the value of an expression is `true` or `false`.

`( cond1 || cond2 )`

is true if either of *`cond1`* or *`cond2`* is true.

|| Example

```
if ((test1==0) || (test2==0))
  cout << "You missed a test!\n";

if ((hw1==0) || (hw2==0))
  cout << "You missed a homework!\n";
```

The ! operator

- The ! operator is a *unary* boolean operator
  - unary means it has only 1 operand.
- ! negates it's operand.
- ! means "not".

`(! condition)`

is true only when `condition` is false

! example

```
bool done = false;
int i=1;

while (! done) {
  cout << "i is " << i << endl;
  i++;
  if (i==100) done=true;
}
```