

Chapter 2

C++ Basics

What is an Identifier?

- An **identifier** is the **name** to denote **labels, types, variables, constants** or **functions**, in a C++ program.
- C++ is a **case-sensitive** language.
 - **Work** is not **work**
- **Identifiers should be descriptive**
 - Using meaningful identifiers is a good programming practice

Identifier

- Identifiers must be unique
- Identifiers cannot be reserved words (keywords)
 - **double main return**
- Identifier must start with a letter or underscore, and be followed by zero or more letters (A-Z, a-z), digits (0-9), or underscores

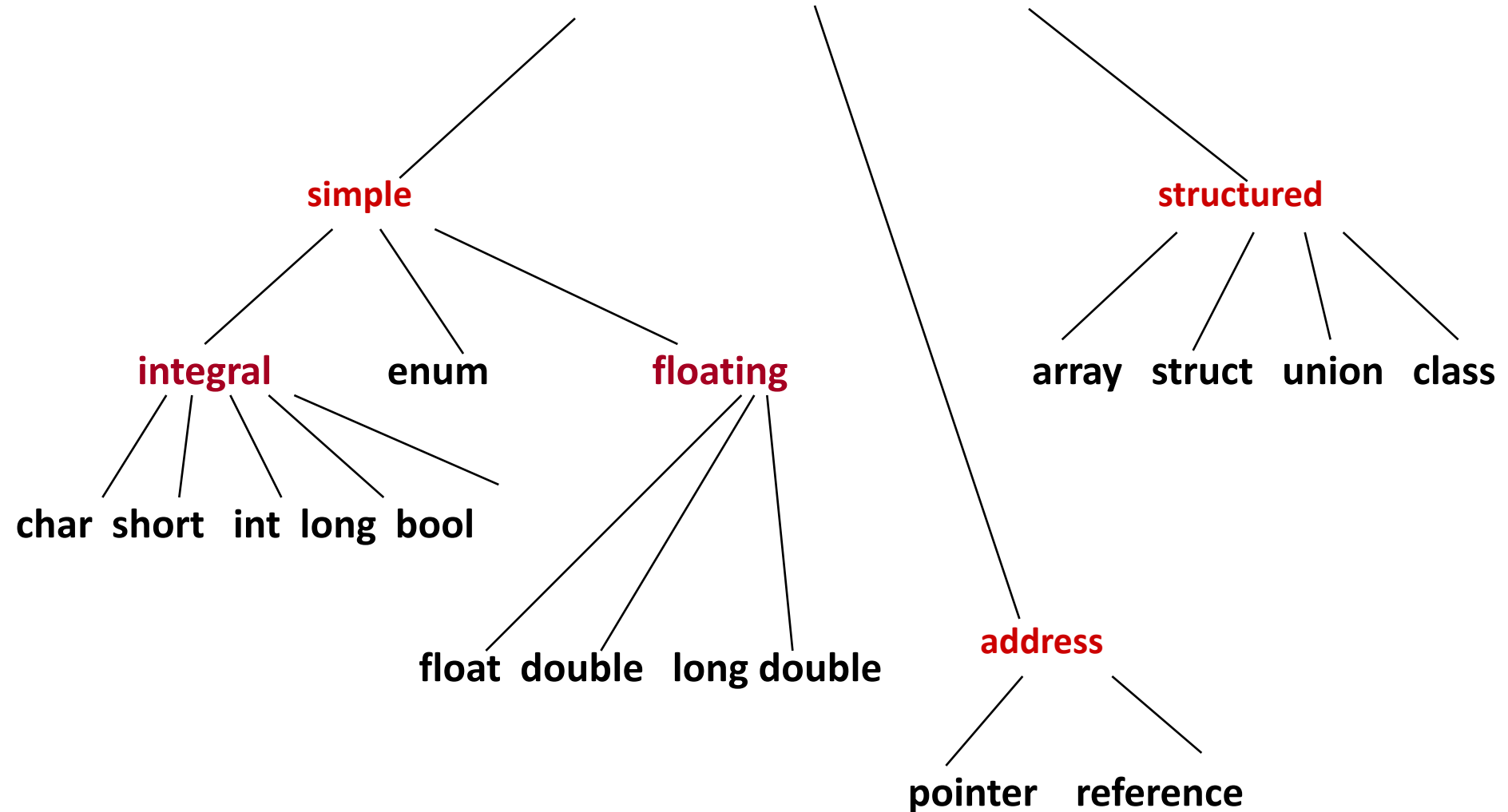
- **VALID**

age_of_dog _taxRateY2K
PrintHeading ageOfHorse

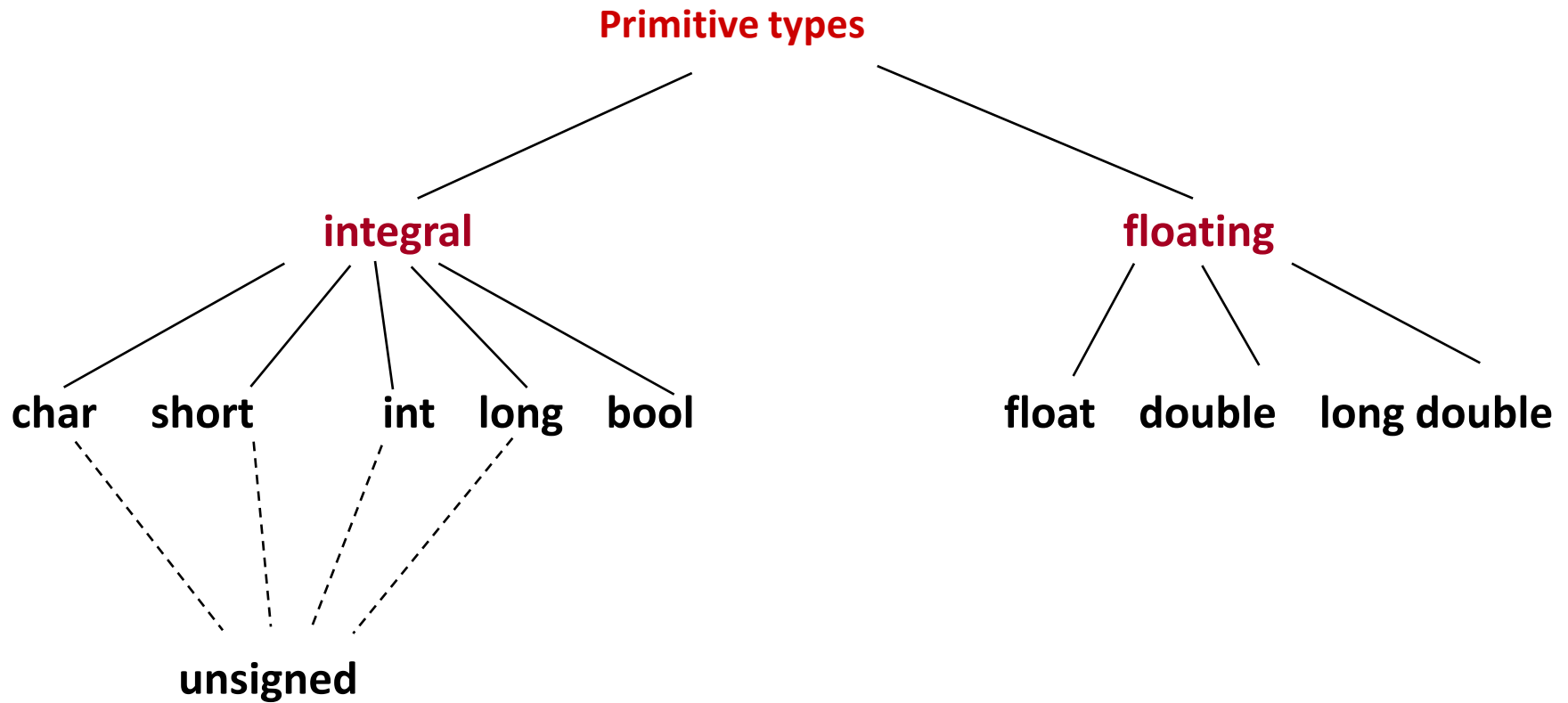
- **NOT VALID**

age# 2000TaxRate Age-Of-Dog main

C++ Data Types



C++ Primitive Data Types



Primitive Data Types in C++

- **Integral Types**
 - represent whole numbers and their negatives
 - declared as `int`, `short`, or `long`
- **Character Types**
 - represent single characters
 - declared as `char`
 - Stored by ASCII values
- **Boolean Type**
 - declared as `bool`
 - has only 2 values `true/false`
 - will not print out directly
- **Floating Types**
 - represent real numbers with a decimal point
 - declared as `float`, or `double`
 - Scientific notation where e (or E) stand for “times 10 to the ” (`.55-e6`)

Samples of C++ Data Values

int sample values

4578

-4578

0

bool values

true

false

float sample values

95.274

95.0

.265

char sample values

'B'

'd'

'4'

'?'

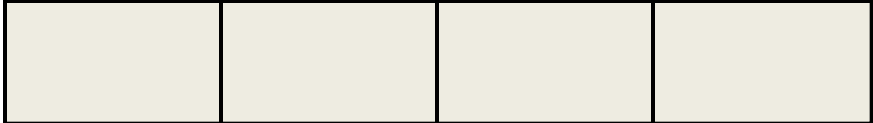
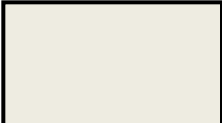

'*'

What is a Variable?

- A **variable** is a memory address where data can be **stored** and **changed**.
- Declaring a variable means specifying both its **name** and its **data type**.

What Does a Variable Declaration Do?

- A declaration tells the compiler to **allocate enough memory** to hold a value of this data type, and to **associate the identifier** with this location.

- `int ageOfDog;` → 
- `char middleInitial;` → 
- `float taxRate;` → 

Variable Declaration

- All **variables** must be declared **before** use.
 - At the top of the program
 - Just before use.
- Commas are used to separate identifiers of the same type.
- Variables can be initialized to a starting value when they are declared

```
int count, age;
```

```
int count = 0;
```

```
int age, count = 0;
```

What is an Expression in C++?

- An **expression** is a valid arrangement of variables, constants, and operators.
- In C++, each **expression** can be evaluated to compute a value of a given type
- In C++, an expression can be:
 - A variable or a constant (count, 100)
 - An operation (a + b, a * 2)
 - Function call (getRectangleArea(2, 4))

Assignment Operator

- An operator to give (assign) a value to a variable.
- Denote as '='
- Only **variable** can be on the left side.
- An **expression** is on the right side.
- Variables keep their assigned values until changed by another **assignment statement** or by **reading in** a new value.

Assignment Operator Syntax

- Variable = Expression
 - First, **expression** on right is **evaluated**.
 - Then the resulting value is **stored** in the memory location of Variable on left.

NOTE: An automatic type coercion occurs **after evaluation but before the value is stored** if the types differ for Expression and Variable

Assignment Operator Mechanism

- Example:

```
int count = 0;
```

A blue rectangular box with a thin black border containing the number 0.

```
int starting;
```

A blue rectangular box with a thin black border containing the text 12345 (garbage).

```
starting = count + 5;
```

- Expression evaluation:

- Get value of **count**: 0

- Add 5 to it.

- Assign to **starting**

A blue rectangular box with a thin black border containing the number 5.

Input and Output

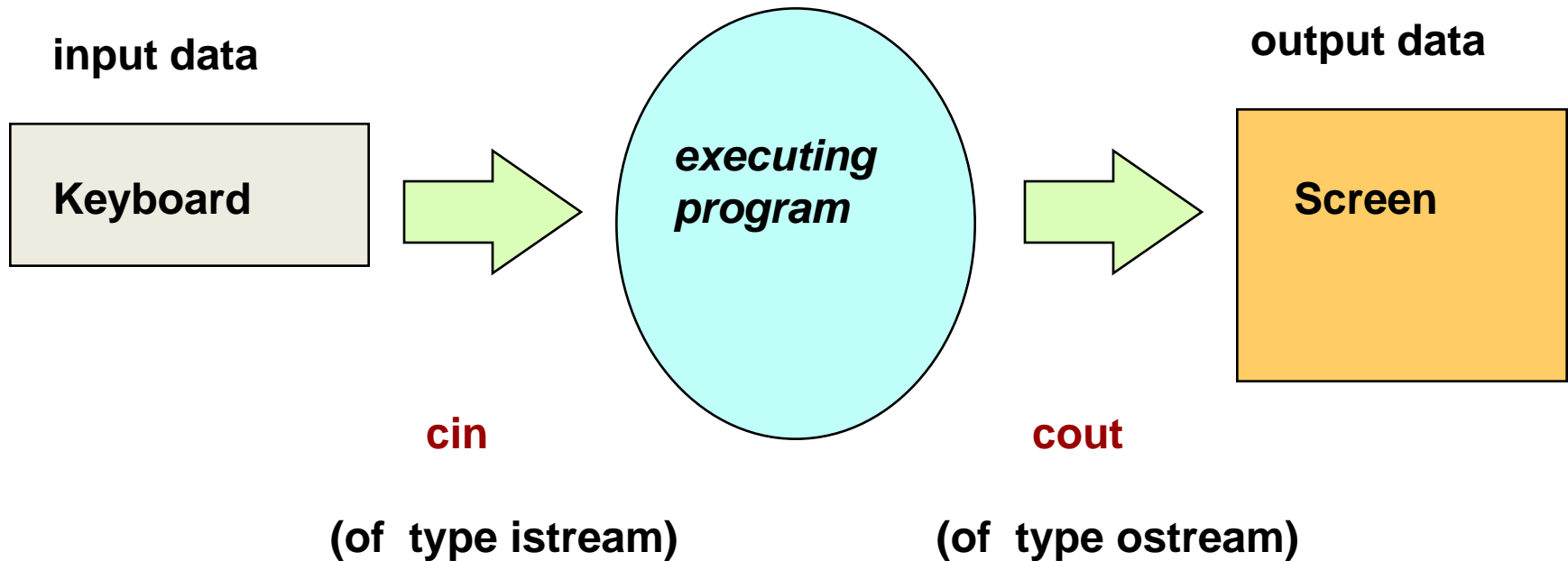
- C++ treats input and output as a **stream** of characters.
- **stream** : sequence of characters (printable or nonprintable)
- The functions to allow standard I/O are in **iostream** header file or **iostream.h**.
- Thus, we start every program with
#include <iostream>
using namespace std;

Include Directives and Namespaces

- **include:** directive copies that file into your program
- **namespace:** a collection of names and their definitions. Allows different namespaces to use the same names without confusion

Keyboard and Screen I/O

```
#include <iostream>
```



Insertion Operator (<<)

- Variable **cout** is predefined to denote an **output stream that goes to the standard output device** (display screen).
- The insertion operator **<<** called “put to” takes 2 operands.
- The left operand is a stream expression, such as **cout**. The right operand is an **expression** of simple type or a **string constant**.

Output Statements

SYNTAX

```
cout << Expression << Expression ... ;
```

cout statements can be linked together using << operator.
These examples yield the same output:

```
cout << "The answer is " ;
```

```
cout << 3 * 4 ;
```

```
cout << "The answer is " << 3 * 4 ;
```

Output Statements (String constant)

- String constants (in double quotes) are to be printed as is, without the quotes.

```
cout<<"Enter the number of candy bars ";
```

OUTPUT: Enter the number of candy bars

- "Enter the number of candy bars " is called a **prompt**.
- All user inputs must be preceded by a **prompt** to tell the user what is expected.
- You must insert **spaces** inside the quotes if you want them in the output.
- Do not put a string in quotes on multiple lines.

Output Statements (Expression)

- All expressions are computed and then outputted.

```
cout << "The answer is " << 3 * 4 ;
```

OUTPUT: The answer is 12

Escape Sequences

- The backslash is called the escape character.
- It tells the compiler that the next character is “escaping” its typical definition and is using its secondary definition.
- Examples:
 - new line: `\n`
 - horizontal tab: `\t`
 - backslash: `\\`
 - double quote `\”`

Newline

- **cout<<"\n"** and **cout<<endl** both are used to insert a blank line.
- Advances the cursor to the start of the next line rather than to the next space.
- Always end the output of all programs with this statement.

Formatting for Decimal Point Numbers

- Typed `float`, or `double`
- Use the three format statements (magic formula) to format to fixed decimal notation.
`cout.setf(ios::fixed);`
`cout.setf(ios::showpoint);`
`cout.precision(2);`
- **setf** “set flag” means that all real output will be formatted according to the function, until changed by either unsetting the flag or a new **setf** command.
- **ios::** means the functions from the **iostream** library

Extraction Operator (>>)

- Variable **cin** is predefined to denote an **input stream from the standard input device** (the keyboard)
- The extraction operator **>>** called “**get from**” takes 2 operands. The left operand is a **stream expression**, such as **cin**--the right operand is a variable of simple type.
- Operator **>>** attempts to **extract** the next item from the input stream **and store** its value in the right operand variable.

Input Statements

SYNTAX

```
cin >> Variable >> Variable ...;
```

cin statements can be linked together using >> operator.

These examples yield the same output:

```
cin >> x;
```

```
cin >> y;
```

```
cin >> x >> y;
```

How Extraction Operator works?

- Input is not entered until user presses **<ENTER>** key.
- Allows backspacing to correct.
- Skips whitespaces (space, tabs, etc.)
- Multiple inputs are stored in the order entered:

cin>>num1>>num2;

User inputs: **3 4**

Assigns **num1** = 3 and **num2** = 4

Numeric Input

- Leading blanks for numbers are ignored.
- If the type is `double`, it will convert `integer` to `double`.
- Keeps reading until blank or <ENTER>.
- Remember to **prompt** for inputs

C++ Data Type String

- A **string** is a **sequence of characters enclosed in double quotes**

- **string** sample values

`"Hello"` `"Year 2000"` `"1234"`

- The empty string (null string) contains no displayed characters and is written as `""`

C++ Data Type String (cont.)

- **string** is not a built-in (standard) type
 - it is a programmer-defined data type
 - it is provided in the C++ standard library
- Need to include the following two lines:
#include <string>
using namespace std;
- **string operations** include
 - comparing 2 string values
 - searching a string for a particular character
 - joining one string to another (concatenation)
 - etc...

Type compatibilities

- **Warning:** If you store values of one type in variable of another type the results can be inconsistent:
 - Can store integers in floating point or in char (assumes ASCII value)
 - bool can be stored as int: (true = nonzero, false = 0)
- Implicit promotion: integers are promoted to doubles
 - double var = 2; // results in var = 2.0**
- On integer and doubles together:
 - Mixed type expressions: Both must be **int** to return **int**, otherwise **float**.

Type compatibilities (Implicit Conversion)

- The compiler tries to be value-preserving.
- General rule: promote up to the first type that can contain the value of the expression.
- Note that representation doesn't change but values can be altered .
- Promotes to the smallest type that can hold both values.
- If assign **float** to **int** will truncate
`int_variable = 2.99; // results in 2 being stored in int_variable`
- If assign **int** to **float** will promote to double:
`double dvar = 2; // results in 2.0 being stored in dvar`

Type compatibilities (Explicit Conversion)

- Casting - forcing conversion by putting (type) in front of variable or expression. Used to insure that result is of desired type.
- Example: If you want to divide two integers and get a real result you must **cast** one to double so that a real divide occurs and store the result in a double.

```
int x=5, y=2; double z; z = static_cast <double>(x)/y; // 2.5
int x=5, y=2; double z; z = (double)x/y; // 2.5
int x=5, y=2; double z; z = static_cast <double>(x/y) ; // 2.0
```

- converts x to double and then does mixed division, not integer divide
- **static_cast<int> (z)** - will truncate z
- **static_cast <int> (x + 0.5)** - will round positive x {use () to cast complete expression)
- Cast division of integers to give real result:

```
int x=5, y=2; double z; z = static_cast <double>(x/y) ; // 2.0
```

Arithmetic Operators

- Operators: +, -, * /
- For **floating numbers**, the result is the same as Math operations.
- Note on **integer** division: the result is an integer. **7/2 is 3.**
- % (remainder or modulo) is the special operator just for integer. It yields an integer as the result. **7%2 is 1.**
- Both / and % can only be used for positive integers.
- Precedence rule is similar to Math.

Arithmetic Expressions

- Arithmetic operations can be used to express the mathematic expression in C++:

$$b^2 - 4ac$$

$$b * b - 4 * a * c$$

$$x(y + z)$$

$$x * (y + z)$$

$$\frac{1}{x^2 + x + 3}$$

$$1 / (x * x + x + 3)$$

$$\frac{a + b}{c - d}$$

$$(a + b) / (c + d)$$

Simple Flow of Control

- Three processes a computer can do:
 - Sequential
 - expressions, insertion** and **extraction** operations
 - Selection (Branching)
 - if** statement, **switch** statement
 - Repetition/Iteration (Loop)
 - while** loop, **do-while** loop, **for** loop

bool Data Type

- Type **bool** is a built-in type consisting of just 2 values, the constants **true** and **false**
- We can declare variables of type **bool**
`bool hasFever; // true if has high temperature`
`bool isSenior; // true if age is at least 55`
- The value 0 represents **false**
- ANY non-zero value represents **true**

Boolean Expression

- Expression that yields **bool** result
- Include:

6 Relational Operators

< <= > >= == !=

3 Logical Operators

! && ||

Relational Operators

are used in boolean expressions of form:

<i>ExpressionA</i>	<i>Operator</i>	<i>ExpressionB</i>
temperature	>	humidity
B * B - 4.0 * A * C	>	0.0
abs (number)	==	35
initial	!=	'Q'

- **Notes:**

- == (equivalency) is **NOT** = (assignment)
- characters are compared alphabetically. However, lowercase letters are higher ASCII value.
- An integer variable can be assigned the result of a logical expression
- You cannot string inequalities together:

Bad Code: `4<x<6` Good Code: `(x > 4) &&(x < 6)`

Relational Operators

```
int x, y ;
```

```
x = 4;
```

```
y = 6;
```

<u>EXPRESSION</u>	<u>VALUE</u>
$x < y$	true
$x + 2 < y$	false
$x \neq y$	true
$x + 3 \geq y$	true
$y == x$	false
$y == x + 2$	true
$y = x + 3$	7
$y = x < 3$	0
$y = x > 3$	1

Logical Operators

are used in boolean expressions of form:

ExpressionA Operator ExpressionB

A || B (true if either A or B or both are true. It is false otherwise)

A && B (true if both A and B are true. It is false otherwise)

or

Operator Expression

!A (true if A is false. It is false if A is true)

Notes:

Highest precedence for NOT, AND and OR are low precedence.

Associate left to right with low precedence. Use parenthesis to override priority or for clarification

- x && y || z will evaluate “x && y” first
- x && (y || z) will evaluate “y || z” first

Logical Operators

```
int age ;  
bool isSenior, hasFever ;  
float temperature ;  
age = 20;  
temperature = 102.0 ;  
isSenior = (age >= 55) ;           // isSenior is false  
hasFever = (temperature > 98.6) ;  // hasFever is true
```

<u>EXPRESSION</u>	<u>VALUE</u>
isSenior && hasFever	false
isSenior hasFever	true
!isSenior	true
!hasFever	false

Precedence Chart

- ++, --, !, - (unary minus), + (unary plus)
- *, /, %
- + (addition), - (subtraction)
- <<, >>
- <, <=, >, >=
- ==, !=
- &&
- ||
- =

Highest



Lowest

Boolean Expression (examples)

taxRate is over 25% and *income* is less than \$20000

temperature is less than or equal to 75 or *humidity* is less than 70%

age is between 21 and 60

age is 21 or 22

Boolean Expression (examples)

`(taxRate > .25) && (income < 20000)`

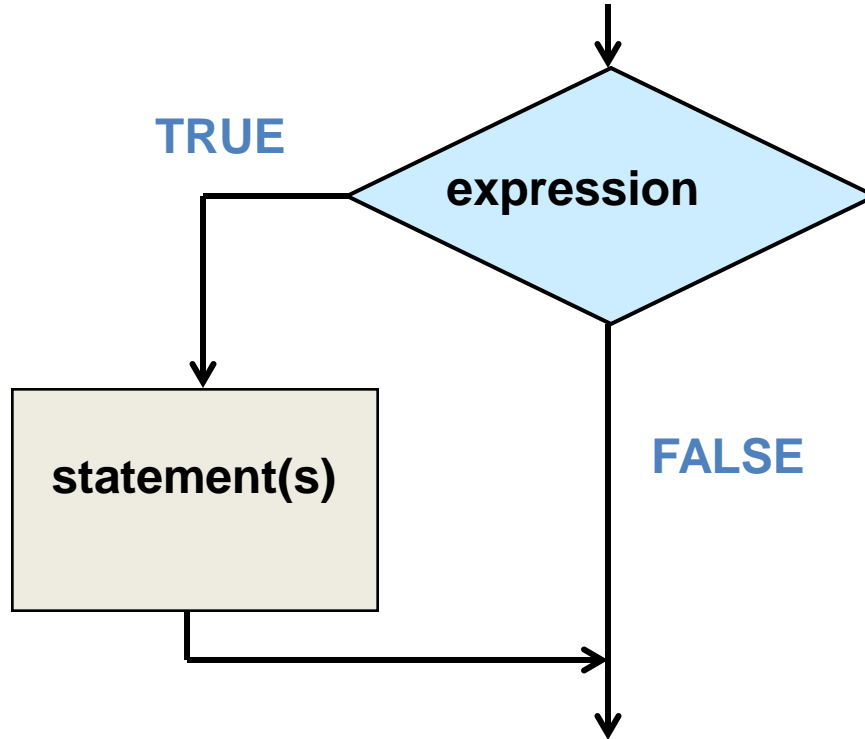
`(temperature <= 75) || (humidity < .70)`

`(age >= 21) && (age <= 60)`

`(age == 21) || (age == 22)`

Simple *if* Statement

- Is a **selection** of whether or not to execute a statement or a block of statement.



Simple *if* Statement Syntax

```
if (Boolean Expression)  
    Statement
```

```
if (Bool-Expr)  
{  
    Statement_1  
    ...  
    Statement_n  
}
```

These are NOT equivalent. Why?

```
if (number == 0 )
{
    cout << "Hmmm ";
    cout << "You entered invalid number.\n";
}
```

When number has value 0, the output will be:

Hmmm You entered invalid number.

When number has value **NOT** 0, there is **NO** output.

```
if (number == 0 )
    cout << "Hmmm ";
    cout << "You entered invalid number.\n";
```

When number has value 0, the output will be:

Hmmm You entered invalid number.

When number has value **NOT** 0, the output will be:

You entered invalid number.

These are equivalent. Why?

```
if (number == 0 )  
{  
    .  
    .  
}
```

Read as:

If *number* is 0

```
if (!number )  
{  
    .  
    .  
}
```

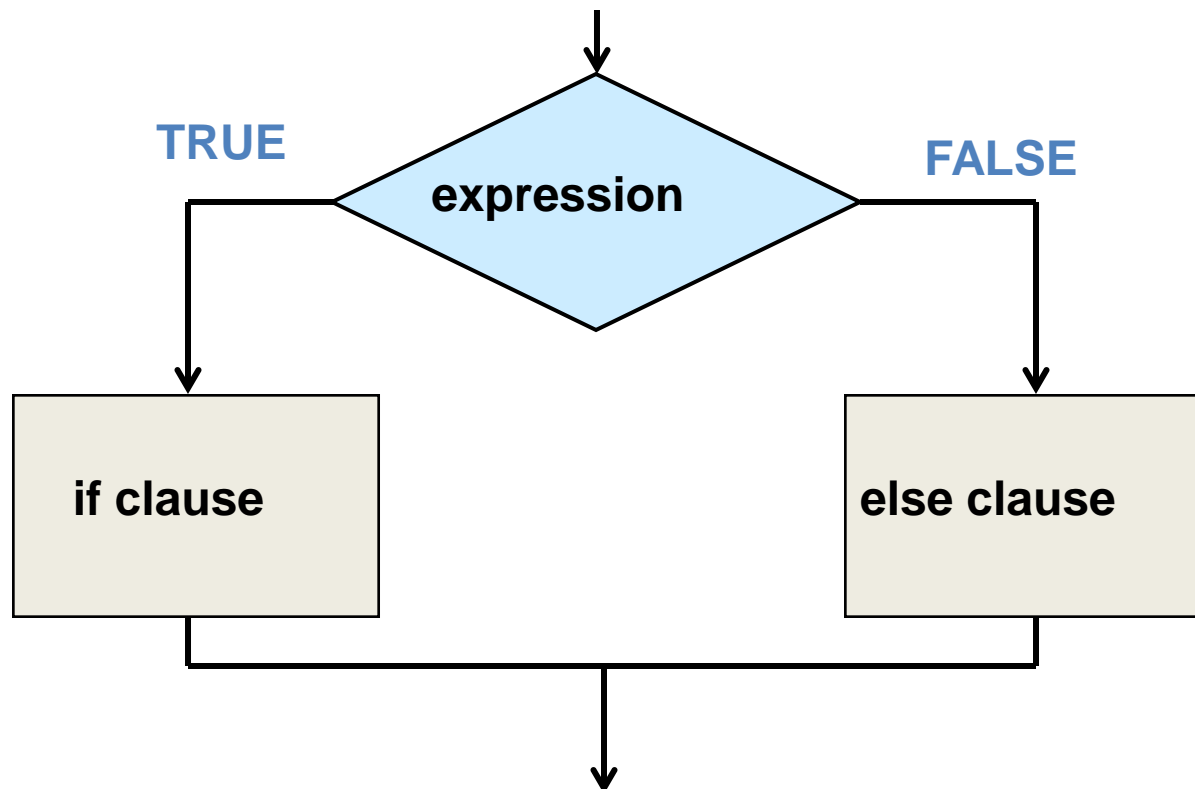
Read as:

If *number* is NOT true
If *number* is false

Each expression is only **true** when number has value **0**.

If-else Statement

- provides **selection** between executing **one of 2 clauses** (the **if** clause or the **else** clause)



Use of blocks

- Denoted by { .. }
- Recommended in controlled structures (if and loop)
- Also called compound statement.

if (*Bool-Expression*)

{

“if clause”



}

else

{

“else clause”



}

Loop

- is a **repetition** control structure.
- causes a **single statement** or **block of statements** to be **executed repeatedly** until a **condition** is met.
- There are 3 kinds of loop in C++:
 - While loop
 - Do-While loop
 - For loop

While Loop

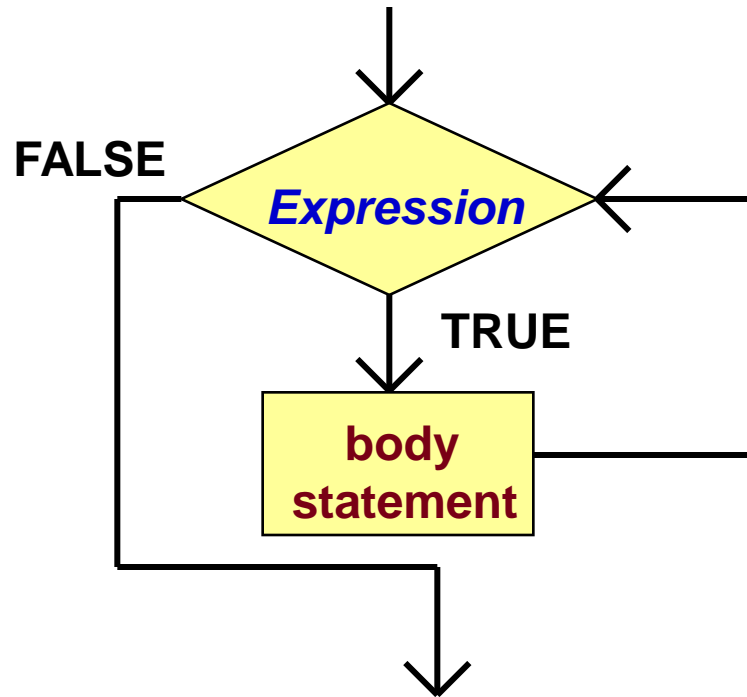
SYNTAX

```
while ( Expression )  
{  
    ... // loop body  
}
```

- No semicolon after the boolean expression
- Loop body can be a single statement, a null statement, or a block.

While Loop Mechanism

- When the expression is tested and found to be **false**, the loop is exited and control passes to the statement which follows the loop body.



- When the expression is tested and found to be **true**, the loop body is executed. Then, the expression is tested again.

While Loop Example

```
int count ;  
count = 0; // initialize LCV  
while (count < 5) // test expression  
{  
    cout << count << " "; // repeated action  
    count = count + 1; // update LCV  
}  
cout << "Done" << endl ;
```

Loop Tracing

```
int count;  
count = 0;  
while (count < 5)  
{  
    cout << count << " ";  
    count = count + 1;  
}  
cout << "Done" << endl;
```

count	Expression	Output
0	true	0
1	true	0 1
2	true	0 1 2
3	true	0 1 2 3
4	true	0 1 2 3 4
5	false	0 1 2 3 4 Done

Increment and Decrement Operators

- Denoted as `++` or `--`
- Mean increase or decrease by 1
- Pre increment/decrement: `++a`, `--a`
 - Increase/decrease by 1 **before** use.
- Post increment/decrement: `a++`, `a--`
 - Increase/decrease by 1 **after** use.
- Pre and Post increment/decrement yield different results when combining with another operation.

Pre and Post

Increment and Decrement

```
int count;  
count = 0;  
while (count < 5)  
{  
    cout << count++ << " ";  
}  
cout << "Done" << endl;
```

```
int count;  
count = 0;  
while (count < 5)  
{  
    cout << ++count << " ";  
}  
cout << "Done" << endl;
```

count	Expression	Output
0	true	0
1	true	0 1
2	true	0 1 2
3	true	0 1 2 3
4	true	0 1 2 3 4
5	false	0 1 2 3 4 Done

count	Expression	Output
0	true	1
1	true	1 2
2	true	1 2 3
3	true	1 2 3 4
4	true	1 2 3 4 5
5	false	1 2 3 4 5 Done

Do-While Loop

SYNTAX

do

{

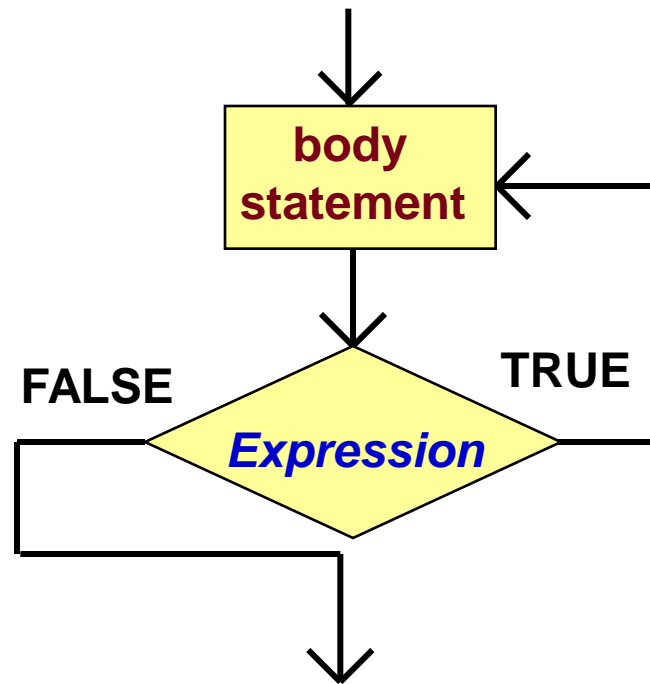
... // loop body

} while (*Expression*);

- Insured that the loop is executed at least **once**
- The LCV is initialized/updated before the end of the loop.
- Boolean expression is tested at the end of the loop.
- There is a semicolon after the boolean expression.

Do-While Loop Mechanism

- The loop body is executed first
- When the expression is tested and found to be **false**, the loop is exited and control passes to the statement which follows the loop body.
- When the expression is tested and found to be **true**, the loop body is executed. Then, the expression is tested again.



Do-While Loop Example

```
int ans;  
do  
{  
    cout << "Choose a number from 1 to 4: "; // repeated action  
    cin >> ans; // LCV is initialized or updated  
} while (ans >= 1 && ans <= 4); // test expression  
cout << "Done";
```

Output	Input	ans	Expression
Choose a number from 1 to 4: 2	2	2	true
Choose a number from 1 to 4: 3	3	3	true
Choose a number from 1 to 4: 1	1	1	true
Choose a number from 1 to 4: 5	5	5	false
Done			

Loop-Controlled Types

Count-controlled: repeat a specified number of times.

Event-driven: some condition within the loop body changes and this causes the repeating to stop.

Sentinel-controlled: using a specific value to end.

Sentinel: a value that cannot occur as valid data.

Ask-before-Continuing: ask users if they want to continue.

Flag-Controlled Loops: use a variable whose value is changed when an event occurs (usually from false to true).

Count-Controlled Loop

- Has a loop control variable (LCV) as a **counter**.
- LCV must be
 - Initialized before start of the loop
 - Tested (boolean expression)
 - Updated

Event-driven loop

```
double salary;
cout << "Enter you salary: ";
cin >> salary;
int years = 0;
while (salary < 50000) {
    salary = salary * 1.02;
    years++;
}
cout << "You need " << years << "years to get to 50K";
```


Sentinel-Controlled

```
do
{
    cout<< "Enter salary, type -1 to exit"; // no one earns negative salary
    cin>>salary;
    // process income
} while (salary > 0);
```

Ask-before-Continuing

```
char ans = 'y';           // LCV is initialized
while (ans == 'Y' || ans == 'y') // test expression
{
    doSomething;         // repeated action
    cout << "Do you want to continue? ";
    cin >> ans;         // LCV is updated
};
```

BREAK statement

allows to exit from any loop.

```
do
{
    cin>>x;
    if (x % 2 ==0)
        break;
} while (x > 0); // exits when an even number is entered
```

CONTINUE Statement

allows you to skip the rest of the loop body and go back to the beginning of the loop.

```
do
{
    cin>>x;
    if (x % 2 == 0)
        continue;
    cout<<x<<endl;
} while (x <100);
//prints out all odd numbers entered less than 100
```

Program Style

- **Indenting:**
 - Separate processes with blank lines
 - Blank lines are also ignored and are used to increase readability.
 - indent statements within statements (loop body)
- **Comments:**
 - `//` tells the computer to ignore this line.
 - for internal documentation. This is done for program clarity and to facilitate program maintenance.

General rules for Comments

- Place a comment at the beginning of every file with the file name, version number, a brief program description, programmer's name.
- Place a descriptive comment after each variable declared.
 - Use a blank line before and after variable declarations
- Place a descriptive comment and a blank line before each subtask.

Constants

- Syntax: `const type identifier = value;`
- Ex: `const double TAX_RATE = 0.08;`
- Convention: use upper case for constant ID.

Why use constants?

- Clarity: Tells the user the significance of the number. There may be the number 0.08 elsewhere in the program, but you know that it doesn't stand for TAXRATE.
- Maintainability. Allows the program to be modified easily.
 - Ex: Program tax compute has **const double TAXRATE=0.0725**. If taxes rise to 8%, programmer only has to change the one line to **const double TAXRATE=0.08**
- Safety: Cannot be altered during program execution