Lecture 1

C++ Basics

Topics

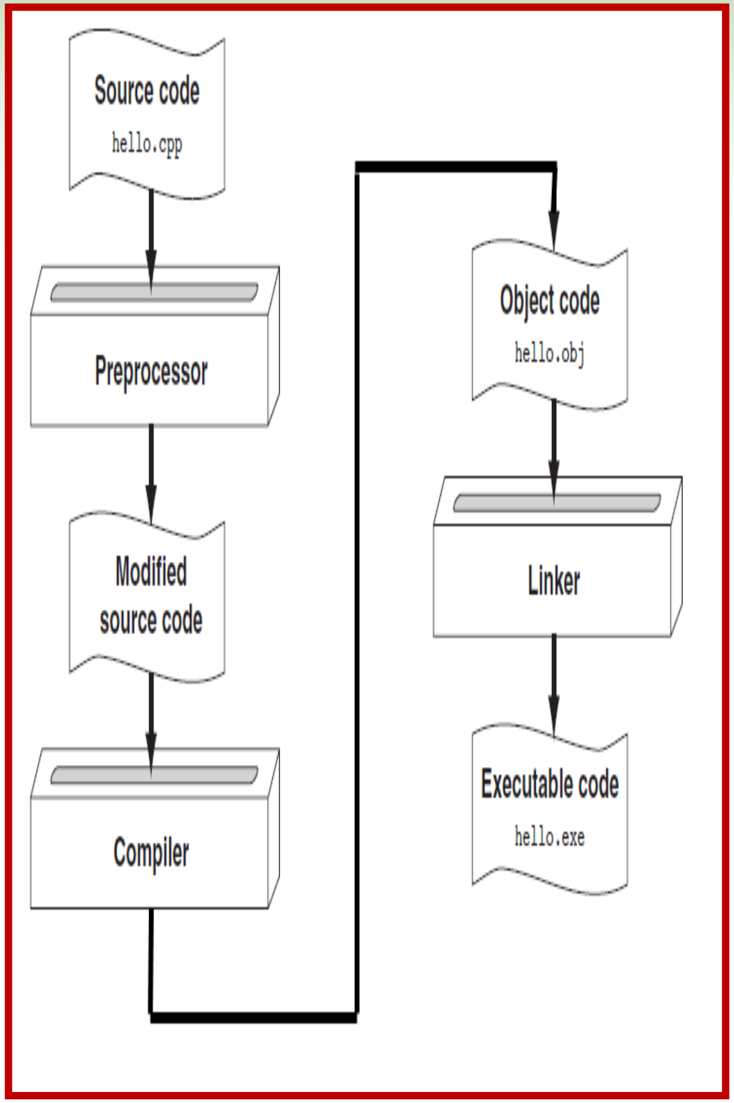
- ❑ C++ basics (the language and basic I/O)
- ❑ Characters and strings
- ❑ C++ control structures (if and switch)
- ❑ C++ Loops
- ❑ C++ Files
- ❑ Arrays (static one and multi dimensional)
- ❑ Vectors
- ❑ functions and passing parameters

Algorithms and Programming

- Program: a set of instructions directing a computer to perform a task
- Programming Language: a special language used to write programs
- Algorithm: a set of well-defined steps to perform a task or to solve a problem (Order is important. Steps must be performed sequentially)
- Types of languages:
  - Low-level: used for communication with computer hardware directly. Not very easy for a person to read.
  - High-level: closer to human language

Write program, compile, and run source code

1. Create a file containing the program with a text editor.
   - ❑ program statements: source code
   - ❑ file: source file
2. Run the preprocessor to convert source file directives to source code program statements.
3. Run the compiler to convert source program statements into machine instructions (machine code), which is stored in an object file.
4. Run the linker to connect hardware-specific library code to machine instructions, producing an executable file.
5. Steps b) through d) are often performed by a single command or button click.
6. Errors that occur at any step will prevent the execution of the following steps.



C++ the language and basic I/O

C++ elements

There are common elements in most programming languages

- ❑ Language elements:
  - – Key Words
  - – Programmer-Defined Identifiers
  - – Operators
  - – Punctuation
  - – Syntax

```
#include <iostream>

using namespace std;

int main()
{
      double   num1 = 5,
               num2, sum;

       num2 = 12;

      sum = num1 + num2;

      cout << "The sum is "
           << sum;

      return 0;
}
```

C++ keywords

- Also known as reserved words
- Have a special meaning in C++
- Can not be used for another purpose
- Written using lowercase letters
- Keywords carried over from C language

auto	const	double	float	int	short	struct	unsigned	break	continue	else
for	long	signed	switch	void	case	default	enum	goto	register	sizeof
typedef	volatile	char	do	extern		if	return	static	union	while

- Keywords new to C++

asm	dynamic _cast	namespace	reinterpret_cast	try	bool	explicit
New	static_cast	typeid	catch	false	operator	template
typename	class	friend	private	this	using	const_cast
inline	public	throw	virtual	delete	mutable	protected
true	wchar_t					

C++ Identifiers, Operators, and Punctuation

Identifiers

- ❑ Names made up by the programmer
- ❑ Not part of the C++ language
- ❑ Used to represent various things, such as variables (memory locations)

```
num2 = 12;
sum = num1 + num2;
```

Punctuation

- ❑ Characters that mark the end of a statement, or that separate items in a list

```
double num1 = 5,
           num2, sum;
num2 = 12;
```

Operators

- ❑ Used to perform operations on data
- ❑ Many types of operators
  - ❑ Arithmetic: **+, -, *, /**
  - ❑ Assignment: **=**
- ❑ Examples in program:

```
num2 = 12;
sum = num1 + num2;
```

Lines vs Statement

In a source file,

- A line is all of the characters entered before a carriage return.
- Blank lines improve the readability of a program.
- Here are four sample lines. Line 3 is blank:

```
1. double num1 = 5, num2, sum;
2. num2 = 12;
3.
4. sum = num1 + num2;
```

In a source file,

- A statement is an instruction to the computer to perform an action.
- A statement may contain keywords, operators, programmer-defined identifiers, and punctuation.
- A statement may fit on one line, or it may occupy multiple lines.
- Here is a single statement that uses two lines:

```
double num1 = 5,
       num2,
       sum;
```

Literals and Variables

- A Literal is a piece of data that is written directly in the source code of the program.

```
// character literal
 'A'

// string literal
"Hello"

// integer literal
  12

//  string literal
"12"

// floating-point literal
3.14
```

- A variable is a named location in computer memory.
- It holds a piece of data.
- A variable must be *defined* before it can be used. Variable definitions indicate the variable name and the type of data that it can hold.
- If a new value is stored in the variable, it replaces the previous value
- The previous value is overwritten and can no longer be retrieved

```
int age;
age = 17;        // Assigns 17 to age
cout << age;     // Displays 17
age = 18;        // Now age is 18
cout << age;     // Displays 18
```

Special Character

Character	Name	Description
//	Double Slash	Begins a comment
#	Pound Sign	Begins preprocessor directive
< >	Open, Close Brackets	Encloses filename used in #include directive
( )	Open, Close Parentheses	Used when naming a function
{ }	Open, Close Braces	Encloses a group of statements
" "	Open, Close Double Quote Marks	Encloses a string of characters
;	Semicolon	Ends a programming statement

- C++ is case-sensitive. Uppercase and lowercase characters are different characters. ‘Main’ is not the same as ‘main’.
- Every { must have a corresponding }, and vice-versa.

Streams

- Stream

  A transfer of information in the form of a sequence of bytes

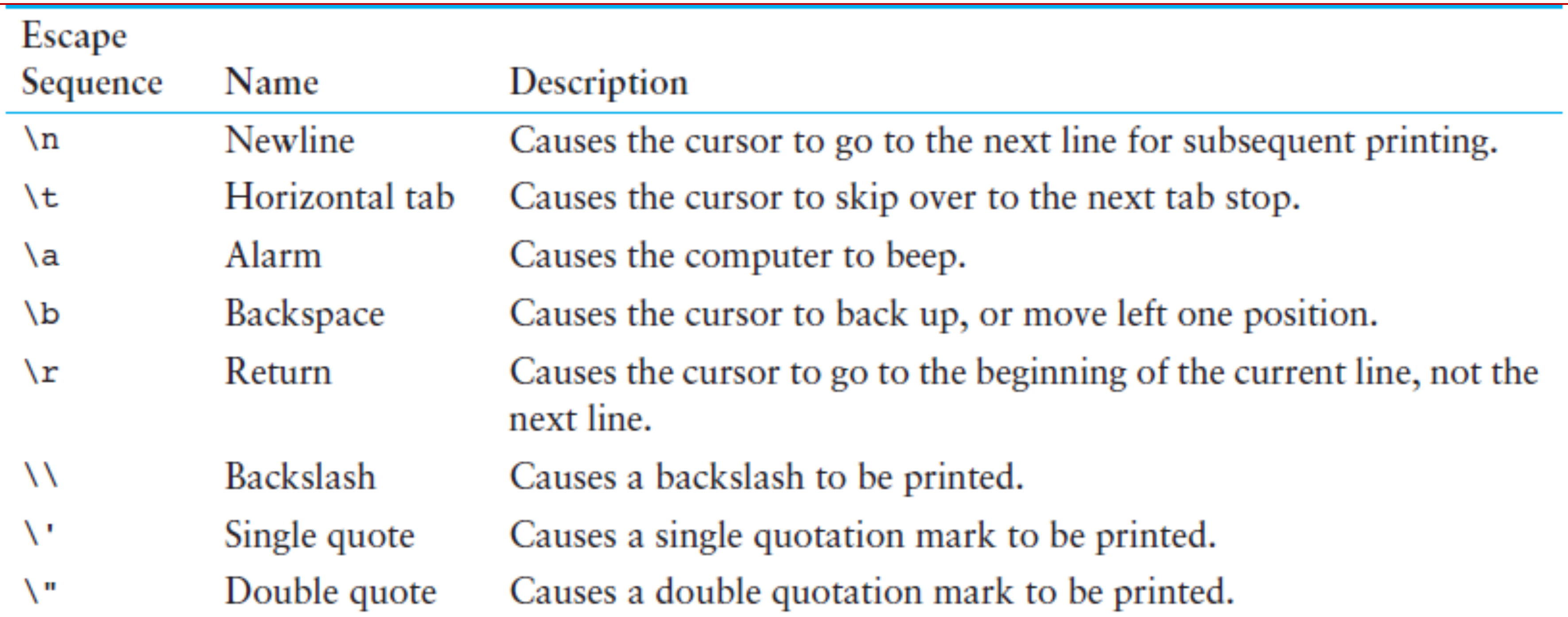
- I/O Operations:
  - Input: A stream that flows from an input device (i.e.: keyboard, disk drive, network connection) to main memory.
  - Output: A stream that flows from main memory to an output device (i.e.: screen, printer, disk drive, network connection).

- iostream library:
  - <iostream.h>: Contains cin, cout, cerr, and clog objects
  - <iomanip.h>: Contains parameterized stream manipulators
  - <fstream.h>: Contains information important to user-controlled file processing operations

The cout object

- The predefined object **cout** is an instance of ostream class.
- The **cout** object is said to be "connected to" the standard output device, which usually is the display screen.
- The **cout** is used in conjunction with the stream insertion operator <<

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello, there!";
   return 0;
}
```



Escape Sequence	Name	Description
`\n`	Newline	Causes the cursor to go to the next line for subsequent printing.
`\t`	Horizontal tab	Causes the cursor to skip over to the next tab stop.
`\a`	Alarm	Causes the computer to beep.
`\b`	Backspace	Causes the cursor to back up, or move left one position.
`\r`	Return	Causes the cursor to go to the beginning of the current line, not the next line.
`\\`	Backslash	Causes a backslash to be printed.
`\'`	Single quote	Causes a single quotation mark to be printed.
`\"`	Double quote	Causes a double quotation mark to be printed.

C++ Data types

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767

Type	Typical Bit Width	Typical Range
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

Determine the size of Data types

- The **sizeof** operator gives the size in number of bytes of any data type or variable

```
double amount;
cout << "A float is stored in " << sizeof(float)
        << " bytes\n";
cout << "Variable amount is stored in ” << sizeof(amount)
        << " bytes\n";
```

Using **auto** in Variable Declarations

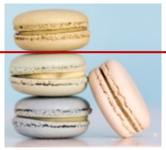
- If you are initializing a variable when it is defined, the **auto** keyword will determine the data type to use based on the initialization value. Introduced in C++ 11.

```
auto length = 12;          // length is an int
auto width = length;       // also int
auto area = 100.0;         // area is a double
```

Scope

- The scope of a variable is that part of the program where the variable may be used
- A variable cannot be used before it is defined

```
#include <iostream>
using namespace std;
int main()
{
  {
    int a = 0;          // scope of the first 'a' begins
    ++a;
    {
        int a = 1;      // scope of the second 'a' begins
                        // scope of the first 'a' is interrupted
        a = 42;         // 'a' is in scope and refers to the second 'a'
    }                   // block ends, scope of the second 'a' ends
                        // scope of the first 'a' resumes
  }                     // block ends, scope of the first 'a' ends
 int b = a;             // Error: name 'a' is not in scope
 return 0;
}
```



Arithmetic Operators

- Used for performing numeric calculations
- C++ has unary, binary, and ternary operators

  unary (1 operand) **-5**

  binary (2 operands) **13 - 7**

  ternary (3 operands) **exp1 ? exp2 : exp3**

SYMBOL	OPERATION	EXAMPLE	`ans`
`+`	**addition**	`ans = 7 + 3;`	`10`
`-`	**subtraction**	`ans = 7 - 3;`	`4`
`*`	**multiplication**	`ans = 7 * 3;`	`21`

`/`	**division**	`ans = 7 / 2;`
		`ans = 7.0 / 2`
			`1`
			`3.5`

`%`	**modulus**	`ans = 7 % 3;`	`1`

Hierarchy of Data Types

- Highest

  ```
  long double
  double
  float
  unsigned long long int
  long long int
  unsigned long int
  long int
  unsigned int
  int
  ```

- Lowest
- Ranked by largest number they can hold

Type Conversion

- Coercion: automatic conversion of an operand to another data type
- Promotion: conversion to a higher type
- Demotion: conversion to a lower type

```
char ch = 'C';
cout << ch << " is stored as "
       << static_cast<int>(ch);
gallons = static_cast<int>(area/500);
avg = static_cast<double>(sum)/count;
intVol1 = (int) volume;       // C-style
intVol2 = int (volume);
```

Overflow

- Occurs when assigning a value that is too large (overflow) or too close to zero (underflow) to be held in a variable
- This occurs with both int and floating-point data types

```
// Create a short int initialized to
// the largest value it can hold

short int num = 32767;

cout << num;      // Displays 32767
num = num + 1;
cout << num;      // Displays -32768
```

Named Constants

- Also called constant variables
- Used for representing constant values with descriptive names

```
const double TAX_RATE = 0.0775;
const int NUM_STATES = 50;
```

- The value of a named constant must be assigned when the variable is defined:

```
const int CLASS_SIZE = 24;
```

- An error occurs if you try to change the value stored in a named constant after it is defined:

```
// This won't work
CLASS_SIZE = CLASS_SIZE + 1;
```

- They make program code more readable by documenting the purpose of the constant in the name:
- They improve accuracy and simplify program maintenance:

Multiple and Combined Assignment

- The assignment operator (**=**) can be used multiple times in an expression

  x = y = z = 5;
- Associates right to left

  x = (y = (z = 5));
- Applies an arithmetic operation to a variable and assigns the result as the new value of that variable
- Operators: **+= -= *= /= %=**
- These are also called compound operators or arithmetic assignment operators
- Example:

  sum += amt; is short for **sum = sum + amt;**

Formatted Output

```
#include <iostream>     // std::cout, std::fixed
#include <iomanip>      // std::setprecision


int main () {
   double f =3.14159;
   std::cout << std::setprecision(5) << f << '\n';
   std::cout << std::setprecision(9) << f << '\n';
   std::cout << std::fixed;
   std::cout << std::setprecision(5) << f << '\n';
   std::cout << std::setprecision(9) << f << '\n';
   return 0;
}
```

```
3.1416
3.14159
3.14159
3.141590000
```

```
#include <iostream>
#include <iomanip>


int main () {
   std::cout << std::setfill ('x') << std::setw (10);
   std::cout << 77 << std::endl;
   return 0;
}
```

```
xxxxxxxx77
```

```
#include <iostream>
#include <iomanip>


int main () {
   std::cout << std::setw(10);
   std::cout << 77 << std::endl;
   return 0;
}
```

```
        77
```

```
const double e = 2.718;
double price = 18.0;
cout << setw(8) << e << endl;                ^^^2.718
cout << left << setw(8) << e << endl;        2.718^^^
cout << setprecision(2);
cout << e << endl;                           2.7
cout << fixed << e << endl;                  2.72
cout << setw(6) << price;                    18.00^
```

Characters and Strings

Characters and Strings

- **char**: holds a single character
- **string**: holds a sequence of characters
- Both can be used in assignment statements
- Both can be displayed with **cout** and **<<**

Strings

- **char**: holds a single character
- **string**: holds a sequence of characters
- Both can be used in assignment statements
- Both can be displayed with **cout** and **<<**

```
#include <iostream>
using namespace std;

int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin >> str;
    cout << "You entered: "
         << str << endl;
    return 0;
}
```

str => MMU

```
#include <iostream>
using namespace std;

int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);

    cout << "You entered: "
         << str << endl;
    return 0;
}
```

str => MMU Cyber

If the user inputs: MMU Cyber

Strings member functions

- **length()** – the number of characters in a string

```
string firstPrez="George Washington";
int size=firstPrez.length();   //17
```

- **length()** includes blank characters
- **length()** does not include the **'\0'** null character that terminates the string
- **assign()** – put repeated characters in a string.
- It can be used for formatting output.

```
string equals;
 equals.assign(80,'=');
 . . .
 cout << equals << endl;
 cout << "Total: " << total << endl;
```

= Assigns a value to a string

```
string words;
words = "Tasty ";
```

+ Joins two strings together

```
string s1 = "hot", s2 = "dog";
string food = s1 + s2;

// food = "hotdog”
```

+= Concatenates a string onto the end of another one

```
words += food;
// words now =
           "Tasty hotdog"
```

C-Strings

- A C-string is stored as an array of characters
- The programmer must indicate the maximum number of characters at definition

```
const int SIZE = 5;
char temp[SIZE] = "Hot";
```

- NULL character (**\0**) is placed after final character to mark the end of the string

H	o	t	\0	

- The programmer must make sure that the array is big enough for desired use. **temp** can hold up to 4 characters plus the **\0**.

- Reading in a C-string

```
const int SIZE = 10;
char Cstr[SIZE];
cin >> Cstr;      // Reads in a C-string with
                  //  no blanks. It will write
                  // past the end of the
                  // array if the input string
                  // is too long.
cin.getline(Cstr, SIZE);
                  // Reads in a C-string that
                  // may contain blanks
                  // Ensures that <= 9
                  // chars are read in.
```

C-Strings Initialization and Assignment

- A C-string can be initialized at the time of its creation, just like a string object

```
const int SIZE = 10;
char month[SIZE] = "April";
```

- However, a C-string cannot later be assigned a value using the = operator; you must use the **strcpy()** function

```
char month[SIZE];
month = "August"        // wrong!
strcpy(month, "August"); //correct
```

- **cin** can be used to put a single word from the keyboard into a C-string
- The programmer must use **cin.getline()** to read an input string that contains spaces
- Note that **cin.getline() ≠ getline()**
- The programmer must indicate the target C-string and maximum number of characters to read:

```
const int SIZE = 25;
char name[SIZE];
cout << "What's your name? ";
cin.getline(name, SIZE);
```

Mathematical Library Functions

- These require **cmath** header file
- They take **double** arguments and return a **double**
- Some commonly used functions

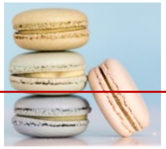
abs	Absolute value
sin	Sine
cos	Cosine
tan	Tangent
sqrt	Square root
log	Natural (e) log
pow	Raise to a power

- Random number - a value that is chosen from a set of values. Each value in the set has an equal likelihood of being chosen.
- Random numbers are used in games and in simulations.
- You have to use the cstdlib header file
- Use time() to generate different seed values each time that a program runs:

```
#include <ctime> //needed for time()
…
unsigned seed = time(0);
srand(seed);
```

- Random numbers can be scaled to a range:

```
int max=6;
int num;
num = rand() % max + 1;
```



Control Structures

Relational Operators

- ❑ Are used to compare numeric and **char** values to determine relative order

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

- ❑ Use this when evaluating an expression that contains multiple relational operators

Operator	**Precedence**
> >= < <=	Highest
== !=	Lowest

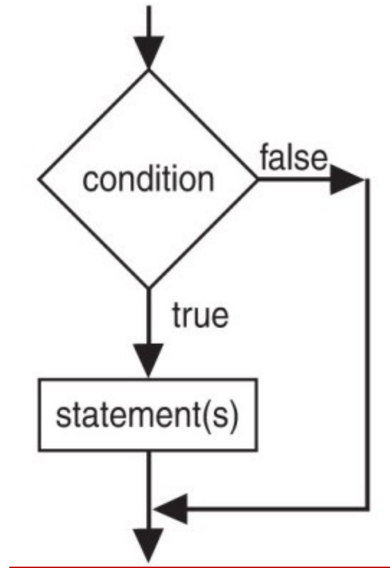
- ❑ Relational expressions are Boolean (*i.e.*, evaluate to **true** or **false)**
- ❑ Examples:

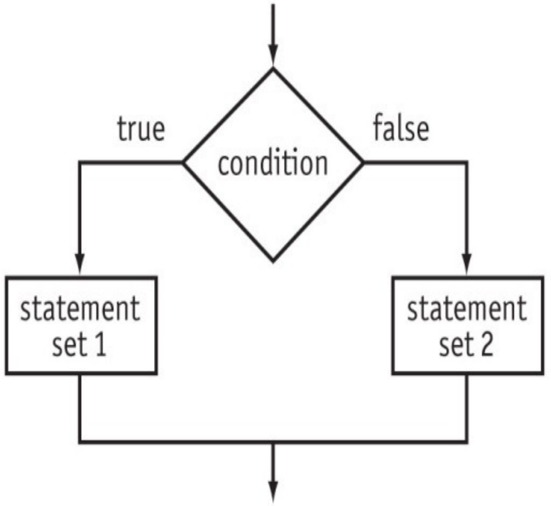
12 > 5 is **true**
7 <= 5 is **false**

if **x** is 10, then

x == 10 is **true**,
x <= 8 is **false,**
x != 8 is **true**, and
x == 8 is **false**

The if statement





```
if (condition)
    {
      statement set 1;
    }
    else
    {
      statement set 2;
    }
```

```
if  (condition)
      {
         statement1;
         statement2;
              …
         statementn;
      }
```

```
if (score >= 60)
  cout << "You passed." << endl;

if (score >= 90)
{
  grade = 'A';
  cout << "Wonderful job!" << endl;
}
```

```
if (score >= 60)
   cout << "You passed.\n";
 else
   cout << "You did not pass.\n";

 if (intRate > 0)
 {
    interest = loanAmt * intRate;
    cout << interest;
 }
 else
       cout << "You owe no interest.\n";
```

Comparing floating point numbers

- ❑ It is difficult to test for equality when working with floating point numbers.
- ❑ It is better to use
  - ❑ greater-than or less-than tests, or
  - ❑ test to see if value is very close to a given value

Nested If Statements

- An **if** statement that is part of the **if** or **else** part of another **if** statement
- This can be used to evaluate > 1 data item or to test > 1 condition

```
if (score < 100)
{
   if (score > 90)
      grade = 'A';
}
```

Logical Operators

Are used to create relational expressions from other relational expressions

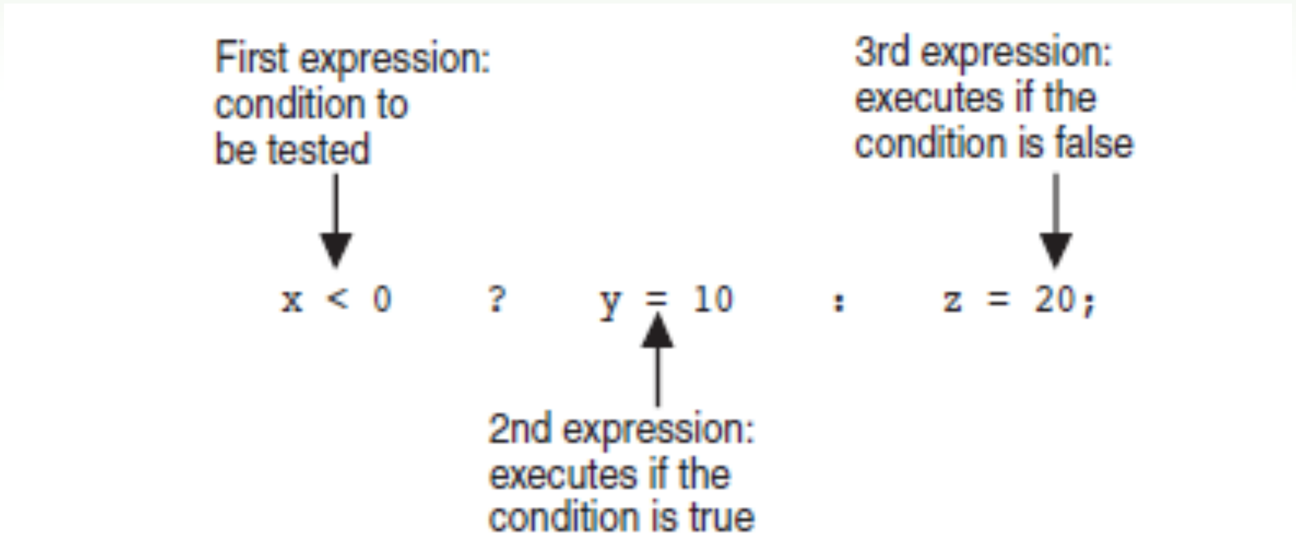
Operator	Meaning	Explanation
&&	**AND**	New relational expression is true if both expressions are true
||	**OR**	New relational expression is true if either expression is true
!	**NOT**	Reverses the value of an expression; true expression becomes false, false expression becomes true

int x = 12, y = 5, z = -4;

`(x > y) && (y > z)`	`true or 1`
`(x > y) && (z > y)`	`false or 0`
`(x <= z) || (y == z)`	`false`
`(x <= z) || (y != z)`	`true`
`!(x >= z)`	`false`

The Conditional Operator

- This can be used to create short **if/else** statements
- Format: **expr ? expr : expr;**


First expression: condition to be tested
3rd expression: executes if the condition is false
x < 0 ? y = 10 : z = 20;
2nd expression: executes if the condition is true

```
int num = 13;
string result= (num%2 ==0) ? "even" : "odd";
cout << num << " is " << result;
```

The switch statement

- Is uses the value of an integer expressiion to determine the statements to execute
- It may sometimes be used instead of **if/else if** statements

```
switch (IntExpression)
{
    case exp1: statement set 1;
    case exp2: statement set 2;
    ...
    case expn: statement set n;
    default:   statement set n+1;
}
```

1) ***IntExpression*** must be an integer variable or a **char**,or an expression that evaluates to an integer value
2) ***exp1*** through ***expn*** must be constant integer type expressions and must be unique in the **switch** statement
3) **default** is optional but recommended

The break Statement

- Is used to stop execution in the current block
- It is also used to exit a **switch** statement
- It is used to execute a single **case** statement without executing statements following it

```
switch (gender)
{
  case 'f': cout << "female";
            break;
  case 'm': cout << "male";
            break;
  default :
      cout << "invalid gender";
}
```

The numerated data type

- ❑ Is a data type created by the programmer
- ❑ Contains a set of named integer constants
- ❑ Format:

  enum *name* {*val1*, *val2*, … *valn*};

- ❑ Examples:

```
enum Fruit {apple, grape, orange};

enum Days {Mon, Tue, Wed, Thur, Fri};
```

- ❑ To define variables, use the enumerated data type name

```
Fruit snack;
Days workDay, vacationDay;
```

- ❑ A variable may contain any valid value for the data type

```
snack = orange;
if (workDay == Wed) …
```

- ❑ Enumerated data type values are associated with integers, starting at 0

```
enum Fruit {apple, grape, orange};
```

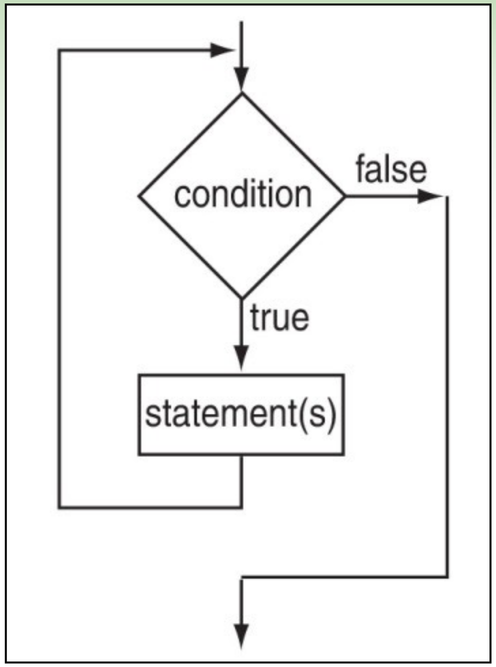
- ❑ You can override the default association

```
enum Fruit {apple = 2, grape = 4,  orange = 5};
```

Loops

The while loop

```
while (condition)
{
    statement(s);
}
```



- ***condition*** is evaluated
  - if it is true, the ***statement(s)*** are executed, and then ***condition*** is evaluated again
  - if it is false, the loop is exited
- An iteration is an execution of the loop body

```
int val = 5;
while (val >= 0)
{   cout << val << "  ";
    val = val - 1;
}
```

produces output:

```
5 4 3 2 1 0
```

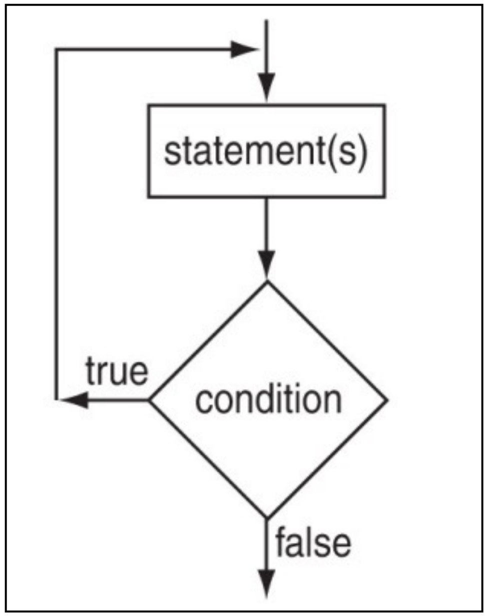
val is called a loop control variable

The do-while loop

- **do-while**: a post test loop (***condition*** is evaluated <u>after</u> the loop executes)

```
do
{
   1 or more statements;
} while (condition);
```

```
do
{
    // code to display menu
    // and perform actions
    cout << "Another choice? (Y/N) ";

} while (choice =='Y'||choice =='y');
```



The condition could be written as

```
(toupper(choice) == 'Y');
```

or as

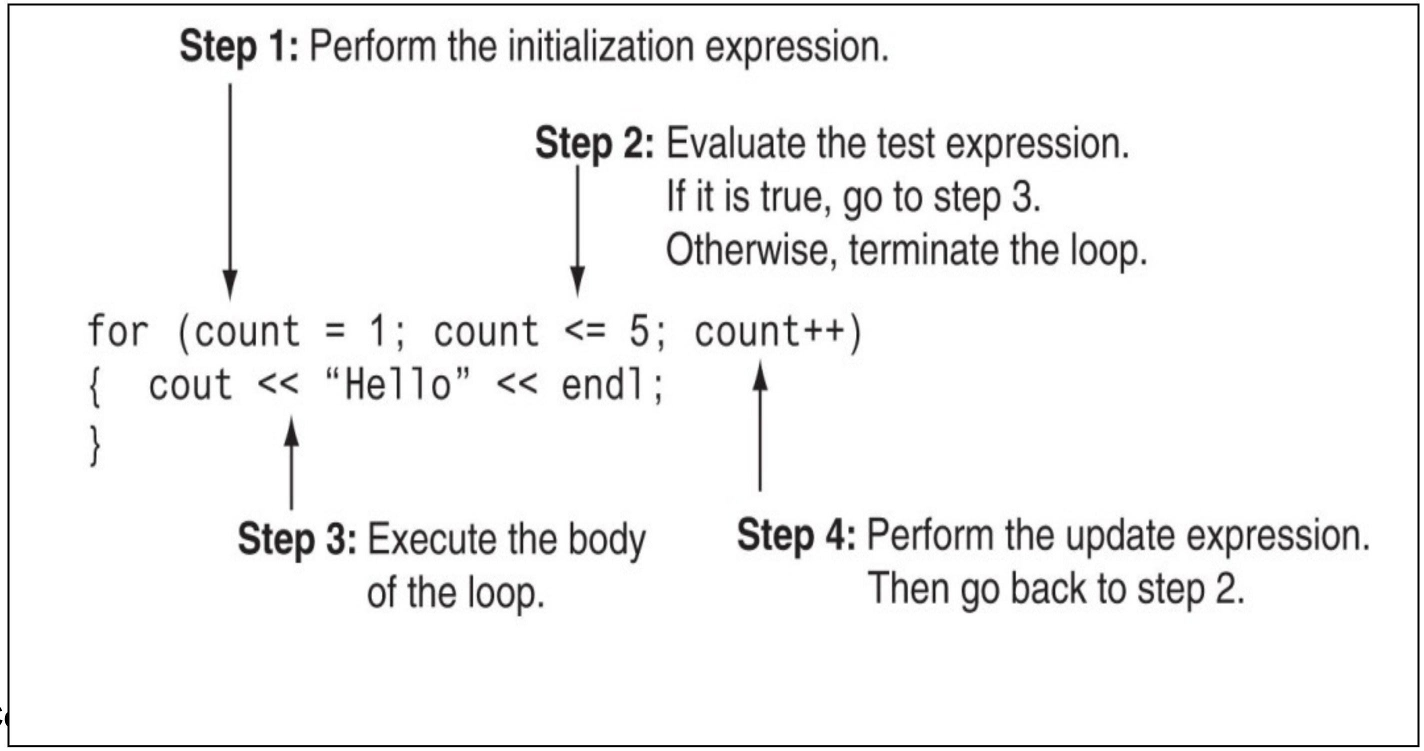
```
(tolower(choice) == 'y');
```

The for loop

```
for( initialization; test; update )
 {
       1 or more statements;
 }
```

```
int sum = 0, num;

for (num = 1; num <= 10; num++)
      sum += num;

cout << "Sum of nums 1 – 10 is "
        << sum << endl;
```

- ❑ It is a pretest loop that executes zero or more times
- ❑ It is useful for a counter-controlled loop



```
for (count = 1; count <= 5; count++)
{   cout << "Hello" << endl;
}
```

Step 1: Perform the initialization expression.
Step 2: Evaluate the test expression. If it is true, go to step 3. Otherwise, terminate the loop.
Step 3: Execute the body of the loop.
Step 4: Perform the update expression. Then go back to step 2.

Nested Loops

- A nested loop is a loop that is inside the body of another loop

```
for (row = 1; row <= 3; row++)
{
     for (col = 1; col <= 3; col++)
   {
    cout << row * col << endl;
   }
}
```

- The inner loop goes through all its iterations for each iteration of the outer loop
- Inner loop complete their iterations complete faster than the outer loop
- Total number of iterations for inner loop is product of number of iterations of the two loops. In previous example, inner loop iterates 9 times in total.

The break and continue keywords

- ❑ **`break`** can be used to terminate the execution of a loop iteration
- ❑ Use it sparingly if at all – it makes code harder to understand
- ❑ When used in an inner loop, **`break`** terminates that loop only and returns to the outer loop

- ❑ You can use continue to go to the end of the loop and prepare for next iteration
  - – while and do-while loops go to the test expression and repeat the loop if test is true
  - – for loop goes to the update step, then tests, and repeats the loop if test condition is true
- ❑ Use continue sparingly – like break, it can make program logic hard to understand

C++ Files

C++ Files

- We can use a file instead of the computer screen for program output
- Files are stored on secondary storage media, such as a disk
- Files allow data to be retained between program executions
- We can later use the file instead of a keyboard for program input
- File Types:
  - Text file – contains information encoded as text, such as letters, digits, and punctuation. It can be viewed with a text editor such as Notepad.
  - Binary file – contains binary (0s and 1s) information that has not been encoded as text. It cannot be viewed with a text editor.
- Ways to use files
  - Sequential access – read the 1st piece of data, read the 2nd piece of data, …, read the last piece of data. To access the n-th piece of data, you have to retrieve the preceding (n-1) pieces first.
  - Random (direct) access – retrieve any piece of data directly, without the need to retrieve preceding data items.

C++ Files

1. Include the **fstream** header file
2. Define a file stream object
   - ❑ **ifstream** for input (read data) from a file (**ifstream inFile;**)
   - ❑ **ofstream** for output (write data) to a file (**ofstream outFile;**)
   - ❑ **fstream** for input from, output to, or both (**fstream file;**)
3. Open the file
   - ❑ Use the **open** member function

     inFile.open("inventory.dat");

     outFile.open("report.txt");
   - ❑ The filename may include drive, path info.
   - ❑ The output file will be created if necessary; an existing output file will be erased first
   - ❑ Input file must exist for **open** to work

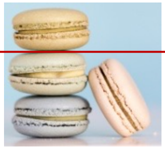
3. Use the file
   - ❑ Can use output file object and << to send data to a file

     outFile << "Inventory report";
   - ❑ Can use input file object and >> to copy data from the file to variables

```
inFile >> partNum;
inFile >> qtyInStock
          >> qtyOnOrder;
```

4. Close the file
   - ❑ Use the close member function

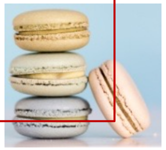
     inFile.close();
     outFile.close();
   - ❑ Don't wait for operating system to close files at program end
   - ❑ There may be limit on number of open files
   - ❑ There may be buffered output data waiting to be sent to a file that could be lost



Text Files

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile;
    myfile.open ("example.txt");
    myfile << "Writing this to a file.\n";
    myfile.close();
    return 0;
}
```

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open()) {
        while ( getline (myfile,line) )
        {
            cout << line << '\n';
        }
        myfile.close();
    }

    else cout << "Unable to open file";

    return 0;
}
```



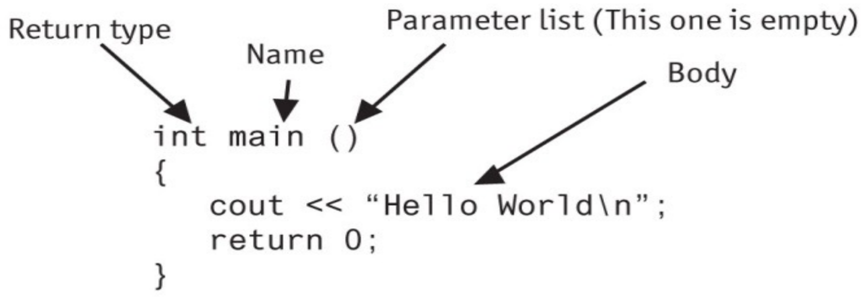
Binary Files

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    streampos size;
    char *memblock;
    ifstream file ("example.bin", ios::in|ios::binary | ios::ate);
    if (file.is_open())  {
        size = file.tellg();
        memblock = new char [size];
        file.seekg (0, ios::beg);
        file.read (memblock, size);
        file.close();
        cout << "the entire file content is in memory";
        delete[] memblock;
    }
    else cout << "Unable to open file";
    return 0;
}
```

Functions

Modular Programming

- **Modular programming**: breaking a program up into smaller, manageable functions or modules. It supports the divide-and-conquer approach to solving a problem.
- **Function**: a collection of statements to perform a specific task
- **Function call**: a statement that causes a function to execute
- **Function definition**: the statements that make up a function
  - name: the name of the function. Function names follow same rules as variable names
  - parameter list: the variables that hold the values that are passed to the function when it is called
  - body: the statements that perform the function's task
  - return type: data type of the value the function returns to the part of the program that called it



```
int main ()
{
    cout << "Hello World\n";
    return 0;
}
```

Parameters, Prototypes, and Function Headings

- For each function argument,
  - the prototype must include the data type of each parameter in its **()**. It may also include the parameter name:

```
void evenOrOdd(int);  or
void evenOrOdd(int num);  // prototype
```

  - the header must include a declaration, with variable type and name, for each parameter in its **()**

```
void evenOrOdd(int num)   //header
```

- The call for the above function could look like this:

```
displayData(height, weight);  // call


void displayData(int h, int w)// header
{
   cout << "Height = " << h << endl;
   cout << "Weight = " << w << endl;
}
```

Passing Data by Value

- Pass by value: when an argument is passed to a function, a copy of its value is placed in the parameter
- The function cannot access the original argument
- Changes made to the parameter in the function do not affect the value of the argument in the calling function

```
#include <iostream>
#include <fstream>
using namespace std;


int fac(int x) {
    int f = 1;
    for (int i=1;i<=x;i++)
        f*=i;
    return f;
}
int main () {
    cout << fac(5) << endl;
    return 0;
}
```

Local and Global Variables

- **local variable**: is defined within a function or a block; accessible only within the function or the block. Parameters are also local variables.
  - Other functions and blocks can define variables with the same name
  - When a function is called, local variables in the calling function are not accessible from within the called function
  - Local variables created when a function begins and are destroyed when the function terminates
- **global variable**: a variable defined outside all functions; it is accessible to all functions within its scope
  - Easy way to share large amounts of data between functions
  - Scope of a global variable is from its point of definition to the program end
- Local variables must be initialized by the programmer
- Global variables are initialized to `0` (numeric) or `NULL` (character) when the variable is defined. These can be overridden with explicit initial values.

Default Arguments

- Values that are passed automatically if arguments are missing from a function call
- Must be a constant or literal declared in the prototype or header (whichever occurs first)

```
void evenOrOdd(int x = 0);
```

- Multi-parameter functions may have default arguments for some or all parameters

```
int getSum(int, int=0, int=0);
```

- If not all parameters to a function have default values, the ones without defaults must be declared first in the parameter list

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // wrong!
```

- When an argument is omitted from a function call, all arguments after it must also be omitted

```
sum = getSum(num1, num2);    // OK
sum = getSum(num1, , num3);  // wrong!
```

Pass by Reference

- This is a mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- It allows the function to modify values stored in the calling environment
- It provides a way for the function to 'return' more than 1 value
- A reference variable is an alias for another variable
- It is defined with an ampersand (**&**) in the prototype and in the header

```
void getDimensions(int&, int&);
```

- Changes made to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters by reference

```
void squareIt(int &);   //prototype
void squareIt(int &num)
{
        num *= num;
}

int localVar = 5;
squareIt(localVar);


// localVar now
// contains 25
```

Overloading Functions

- The signature of a function is the function name and the data types of the parameters, in order.
- Overloaded functions are two or more functions that have the same name but different signatures
- This can be used to create functions that perform the same task but take different parameter types or a different number of parameters

- If a program has these overloaded functions,

```
void getDimensions(int);                  // 1
void getDimensions(int, int);             // 2
void getDimensions(int, float);           // 3
void getDimensions(double, double);       // 4
```

- then the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);               // 1
getDimensions(length, width);        // 2
getDimensions(length, height);       // 3
getDimensions(height, base);         // 4
```

Arrays

Arrays

- Array: a variable that can store multiple values of the same type
- The values are stored in consecutive memory locations
- It is declared using **[ ]** operator

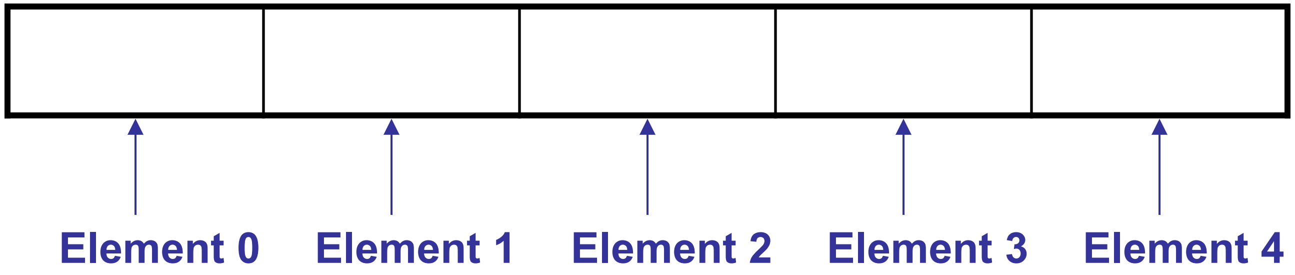
```
const int ISIZE = 5;
int tests[ISIZE];
```

- The definition

```
int tests[ISIZE];  // ISIZE is 5
```

- allocates the following memory

```
tests[0] = 79;

cout << tests[0];

cin  >> tests[1];

tests[4] = tests[0] + tests[1];

cout << tests;
  // illegal due to
  // missing subscript
```



Array Initialization

- An array can be initialized during program execution with assignment statements

```
tests[0] = 79;
tests[1] = 82; // etc.
```

- It can be initialized at array definition with an initialization list

```
const int ISIZE = 5;
int tests[ISIZE] = {79,82,91,77,84};
```

- If array is initialized at definition with fewer values than the size declarator of the array, the remaining elements will be set to 0 or the empty string

```
int tests[ISIZE] = {79, 82};
```

79	82	0	0	0

- Initial values are used in order; you cannot skip over elements to initialize a noncontiguous range
- You cannot have more values in the initialization list than the declared size of the array

Implicit Array Sizing

- C++ can determine the array size by the size of the initialization list

```
short quizzes[]={12,17,15,11};
```

12	17	15	11

- You must use either array size declarator or an initialization list when the array is defined
- You can initialize a variable at definition time using a functional notation

```
int length(12);  // same result as
                 // int length = 12;
```

- In C++ 11 and higher, you can also do this:

```
int length{12};
```

- The second approach checks the argument to ensure that it matches the data type of the variable, and will generate a compiler error if not

Range Based for Loop

- This is a loop that can simplify array processing.
- It uses a variable that will hold a different array element for each iteration
- Format:

    for (data_type var : array)

    statement;

- **data_type** must be the type of the array elements, or a type that the array elements can be automatically converted to
- **var** will hold the value of successive array elements as the loop iterates. Each array element is processed in the loop
- **statement;** can be a single statement or a block of statements enclosed in { }

```
// sum the elements of an array
int [ ] grades = {68,84,75};
int sum = 0;
for (int score : grades)
              sum += score;
```

```
// modify the contents of an array
const int ISIZE = 3;
int [ISIZE] grades;
for (int &score : grades) {
   cout << "Enter a score: ";
   cin >> score;
}
```

Copying One Array to Another

- You cannot copy with an assignment statement:

```
tests2 = tests;  //won't work
```

- You must instead use a loop to copy element-by-element:

```
for (int indx=0; indx < ISIZE; indx++)
    tests2[indx] = tests[indx];
```

- Like copying, you cannot compare two arrays in a single expression:

```
if (tests2 == tests)
```

- You can use a while loop with a **bool** variable:

```
bool areEqual=true;
int indx=0;
while (areEqual && indx < ISIZE)
{
   if(tests[indx] != tests2[indx]
      areEqual = false;
   indx++;
}
```

The typedef Statement

- Creates an alias for a simple or structured data type
- Format:

  typedef *existingType newName*;
- Example:

```
typedef unsigned int Uint;
Uint tests[ISIZE];    // array of
                      // unsigned ints
```

- It can be used to make code more readable
- Can be used to create an alias for an array of a particular type

```
// Define yearArray as a data type
// that is an array of 12 ints

typedef int yearArray[MONTHS];

// Create two of these arrays


yearArray highTemps, lowTemps;
```

Arrays as Function Arguments

- Passing a single array element to a function is no different than passing a regular variable of that data type
- The function does not need to know that the value it receives is coming from an array
- To define a function that has an array parameter, use empty [ ] to indicate the array in the prototype and header
- To pass an array to a function, just use the array name

```
displayValue(score[i]);        // call

void displayValue(int item) // header
{
        cout << item << endl;
}
```

```
void showScores(int []);
// Function prototype
void showScores(int tests[])
// Function header
showScores(tests);
// Function call
```

```
showScores(tests, 5);                  // call
void showScores(int[], int);          // prototype
void showScores(int A[], int size)     // header
```

Two Dimensional Arrays

- You can define one array for multiple sets of data
- It is like a table in a spreadsheet
- Use two size declarators in definition

```
int exams[4][3];
```

rows	columns		
	exams[0][0]	exams[0][1]	exams[0][2]
	exams[1][0]	exams[1][1]	exams[1][2]
	exams[2][0]	exams[2][1]	exams[2][2]
	exams[3][0]	exams[3][1]	exams[3][2]

- Two-dimensional arrays are initialized row-by-row

```
int exams[2][2] = { {84, 78}, {92, 97} };
```

- Can omit inner **{ }**

Passing a Two Dimensional Array to a Function

- Use the array name and the number of columns as arguments in the function call

```
getExams(exams, 2);
```

- Use empty **[]** for row and a size declarator for the number of columns in the prototype and header

```
// Prototype, where NUM_COLS is 2
void getExams(int[][NUM_COLS], int);
// Header
void getExams
        (int exams[][NUM_COLS], int rows)
```

Vectors

- A vector holds a set of elements, like a one-dimensional array
- It has a flexible number of elements. It can grow and shrink
  - There is no need to specify the size when defined
  - Space is automatically added as needed
- Defined in the Standard Template Library (STL)
  - Covered in a later chapter
- Must include vector header file to use vectors (#include <vector>)\It can hold values of any data type, specified when the vector is defined

```
vector<int> scores;
vector<double> volumes;
```

- You can specify initial size if desired

```
vector<int> scores(24);
```

- Use [ ] to access individual elements

Defining Vectors

- Define a vector of integers (starts with 0 elements)

```
vector<int> scores;
```

- Define **int** vector with initial size 30 elements

```
vector<int> scores(30);
```

- Define 20-element **int** vector and initialize all elements to 0

```
vector<int> scores(20, 0);
```

- Define **int** vector initialized to size and contents of vector **finals**

```
vector<int> scores(finals);
```

- C++ 11 supports vector definitions that use an initialization list

```
vector<int> scores {88, 67, 79, 84};
```

- Note: no = operator between the vector name and the initialization list

- A range-based for loop can be used to access the elements of a vector

```
for (int test : scores)
   cout << test << " ";
```

Growing a Vector's Size

- Use the **push_back** member function to add an element to a full vector or to a vector that had no defined size

```
// Add a new element holding a 75
scores.push_back(75);
```

- Use the **size** member function to determine the number of elements currently in a vector

```
howbig = scores.size();
```

- Use the **pop_back** member function to remove the last element from a vector

```
scores.pop_back();
```

- Note: **pop_back** removes the last element but does not return it
- To remove all of the elements from a vector, use the **clear** member function

```
scores.clear();
```

- To determine if a vector is empty, use **empty** member function

```
while (!scores.empty()) ...
```