C++ Beginning Basic Summary Sheet

Problem Solving with C++ by Walter Savitch

C++ Program Structure

```
#include <directives>
// no spaces around # or within <>
const variable DECLARATIONS;
struct Structure_Tag
{
    type member_name ...
};
function_declarations();
int main()
{
    using namespace std;         // see p. 228-29
    ...                          // do something
    return 0;
}
function_definitions()
{
    using namespace std;
    ...
}
```

General

Variable Declarations:
- Variables must be declared before use.
- Declaring the variable tells the compiler what *type* of data to store.
- Syntax: *type_name variable_name1, variable_name2;*
- Examples:
```
int count, rate;
double distance;
```

Initializing Variables
- Give the variable a value when you declare it.
- Initializing gives the variable its *first* *value*.
- Syntax (either or):
  type_name var_name1 = value1, var_name2 = value2;
  type_name var_name1(value1), var_name2(value2);
- Examples:
```
int count = 100, rate = 10;
double distance(24.2);
```

Constants: variables that cannot be changed
const type_name VARIABLE_NAME = constant_value;
```
const double PI(3.14159);
```

Raw String Literals:
- Convenient if you have multiple escape sequences in a statment.
- Example:
```
cout << R"(c:\files\path\here)";
// output: c:\files\path\here
```

Quotes:
Single: used around type char (e.g. 'a')
Double: used around strings (e.g., cout << "This";)

Compile in Linux (g++)

Single File C++ Programs

Format1:
```
g++ filename.cpp // default output: a.out
Execute:  ./a.out
```
Format2:
```
g++ filename.cpp -o outputFilename
Execute:  ./outputFilename
```

Operators

Assignment Operator Shortcuts
```
count += 2;         =>  count = count + 2;
count -+ 2;         =>  count = count - 2;
count *= 2;         =>  count = count * 2;
count /= 2;         =>  count = count / 2;
count %= 2;         =>  count = count % 2;
count *= v1 + v2;   =>  count = count * (v1 + v2);
```

Increment / Decrement Operators (use only w/variables)
```
var++ => increment 1  AFTER  the value is returned
++var => increment 1  BEFORE the value is returned
var-- => decrement 1  AFTER  the value is returned
--var => decrement 1  BEFORE the value is returned
```

Comparison Operators (boolean results)

==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Boolean Operators

&&	and	**0 = false**
||	or	**non-zero = true**
!	not (avoid using this one)	

Data Types

Commonly Used Types

int	integer
double	integer plus decimal
char	one character
bool	true (non-zero) or false (0)
enum	list of constants (use enum classes)

Type Casting: Changing Data Types
```
static_cast<new_type>(expression);
new_double = static_cast<double>(var_of_type_int);
```

Variables

Identifiers: Variable *names* are called *identifier*.

1. Automatic (i.e., ordinary): all dynamic properties are controlled automatically.

2. Dynamic: created and destroyed while program is running (see pointers).

3. Global: declared outside any function (including main); seldom needed.

Flow of Control

Break
use "break" to exit a loop before it ends normally.

Exit
use "exit(0)" to exit the entire program altogether.

If... Else If... Else...
```
if (bool exp)      // do something;
else if (bool exp) // do something else;
else               // do something else;
```
Need { } if more than one line

Switch
Leave out "break" to apply one statement sequence to more than one case.
```
switch (controlling_statement)
{
    case constant_1:
        statement_sequence_1;
    case constant_2:
        statement_sequence_2;
        break;
    ...
    case constant_n:
        statement_sequence_n;
        break;
    default:
        default_statement_sequence;
}
```
Controlling statement can be: bool, enum, int, char

While
```
while (bool exp) {
   // do something
   // Compound statements require brackets
}
```

Do-While
```
do {
    // do something
} while (bool exp);     // remember the semicolon
```

For
Syntax: *for (initialize; bool_eval; update)*
Example:
```
for ( int n = 1; n <= 10; n++ )
{
    // do something in each iteration
}
```

Range-Based For Loop (simplies array iteration)
Simple loop:
```
int my_array[] = {2, 3, 4, 5, 10};
for ( int x : my_array )
    cout << x;              // output: 234510
```
Pass-by-reference to change the element values:
```
int my_array[] = {2, 3, 4, 5, 10};
for ( int& x : my_array )
    x++;
for ( auto x : my_array )
    // auto: automatically determine type
    cout << x;              // output: 345611
```

FUNCTIONS: General

1. function declaration:

```
type_returned function_name(para_type para_list);
// Precondition: what the function needs
// Postcondition: what the function does/returns
```

2. function call:

```
function_name(argument_list);
```

3. function definition:

```
type_returned function_name(para_type para_list)
  {
     do something;
     return something;
  }
```

Void Functions: return no value. Use keyword "void" and no value returned.

```
void function_name(para_type para_list)
{
    do something;
    return; // return is optional in void functions
}
```

Overloading: give two or more different definitions to the same name.

1. Each definition must have a different NUMBER of parameters, AND/OR
2. Each definition must have different parameter TYPES.

FUNCTIONS: Parameters

Parameters: these are the "placeholders" in the function declaration and definition.

- Can be call-by-value, call-by-reference, or mixed.
- Call-by-reference is determined by the addition of an "**&**".

Arguments: these are what the user passes to the function in place of the parameters.

Call-By-Value: (*type parameter*) | (*int num1*)

- When called, the VALUES of the arguments are substituted for the parameters
- Can use variables (containing values) or literals (values themselves)

Call-By-Reference: (*type& parameter*)

- Example:
```
void get_input (double& f_variable)
       cin >> f_variable;            }
```
- Indicated with an "&" appended to the parameter type name.
- When called, the argument VARIABLES are substituted for the parameters.
- The reference to the memory address of the variable is passed to the function
  - ~ The memory address of the argument is passed to the parameter.
  - ~ Therefore the parameter (as a variable) in the function has the same address.
- Therefore, the arguments MUST be VARIABLES.
- The function code, then, can change the argument variable (change value at memory address).
- You must include the **&** in both the declaration and the definition, not the call.

FUNCTIONS: Usage

1. Use **call-by-reference** when you need to CHANGE the VALUE of a passed-in VARIABLE.
2. Use **call-by-value** in all other cases (when you just need to pass in a value).

FUNCTIONS: Default Arguments

You can assign a default value to function parameters:

```
void new_line(istream& in_stream = cin);
```

If you mix & match parameters, ALL the parameters with defaults must go at the END of the parameter list.

FUNCTIONS: Streams as Arguments

Streams can be arguments to functions but they must be **CALL-BY-REFERENCE**; they cannot be call-by-value.

I/O Member Functions

All data is input and output as CHARACTER data.
- I/O => cin, cout, ifstream, ofstream.
- All numbers are regarded as characters & converted.
- get() and put() can side-step this conversion...

.get()

Reads in one char of input & stores it in a char var.
- Every input stream (cin, ifstream) has .get()
- With .get() you get EVERY character input, including spaces and newlines (\n).
- Takes one argument: variable to type char to receive input from stream

```
char c1, c2, c3, c4; OR:  char symbol;
cin.get(c1);              do {
cin.get(c2);                  cin.get(symbol);
cin.get(c3);                  cout << symbol;
cin.get(c3);              } while ( symbol != '\n' );
```

.getline()

Reads entire line of input into a C-String variable.

cin.getline(Cstring_var, max_char + 1)

Reads in until end of line or end of max_char in 2nd parameter (+1 for null char at end: '\0').

.put()

Outputs one character to the indicated output stream
- Takes one argument that should be a char expression
- Examples: cout.put(next_symbol); *OR* cout.put('a');

.putback()

Places one character back in the input stream
- Good for reading up to a certain character (like a blank), stop processing and put flag character back.
- Example:
```
file_in.get(next);
while ( next != ' ' ) {
    file_out.put(next);
    file_in.get(next);
}
file_in.putback(next);
```

.

Predefined CHARACTER Functions

Functions that operate on characters. Must include the header... #include <cctype>

toupper(char_expr)*	returns upper case
tolower(char_expr)*	returns lower case
isupper(char_expr)	true if uppercase, else false
islower(char_expr)	true if lowercase, else false
isalpha(char_expr)	true if alpha, else false
isdigit(char_expr)	true if digit, else false
isspace(char_expr)	true if whitespace, else false

* *toupper* and *tolower* return NUMERIC VALUES that corresponde to the letter.
- You must indicate expressly that you want a char returned, not a number.
- One way to do that is assign the return value to a variable of type char:
```
char c = toupper('a');
```
- OR type-cast the output:
```
cout << static_cast<char>(toupper('a'));
```

Predefined MATH Functions

sqrt — **square root**

```
#include <cmath>
using namespace std;
...
x = sqrt(y);
```

rand — **random number generator**

```
#include <cstdlib>
#include <ctime>
...
srand( time(0) ); // seed only once!
int num1 = ( rand() % 6) + 1 ) // random, 1-6
int num2 = ( rand() % 9) + 1 ) // random, 1-10
```

abs — **absolute value for int**

```
#include <cstdlib>
using namespace std;
...
int abs_value = abs(-7)   // value: 7
int abs_value = abs(7)    // value: 7
```

ceil — **ceiling (round up)**

```
#include <cmath>
using namespace std;
...
double ceil_value = ceil(3.2) // value: 4.0
double ceil_value = ceil(3.9) // value: 4.0
```

floor — **floor (round down)**

```
#include <cmath>
using namespace std;
...
double ceil_value = ceil(3.2) // value: 3.0
double ceil_value = ceil(3.9) // value: 3.0
```

File I/O Includes

```
#include <iostream> // for cin and cout
#include <fstream>  // file I/O (ifstream & ofstream)
#include <cstdlib>  // for exit()
using namespace std;
```

I/O Streams

A stream is a variable, an OBJECT created from a CLASS

- You have to **(1) declare** the stream and
  (2) connect the stream to a file.
- The value of the stream "variable" is the file it's connect to.

TYPES: Definied in the *fstream* library

ifstream - the type for input-file stream variables
ofstream - the type for output-file stream variables

[1] DECLARING STREAMS:
(creates objects of class ifstream & ofstream)

```
ifstream in_stream;  // declares variable in_stream
                        of type ifstream

ofstream out_stream; // declares variable out_stream
                        of type ofstream
```

[2] CONNECT: stream vars must be "connected" to files

```
in_stream.open("infile.dat");
// connects in_stream to file infile.dat

out_stream.open("outfile.dat");
// connects out_stream to outfile.dat
```

Note: APPEND to an existing file by adding a 2nd arg:

```
out_stream.open("outfile.dat", ios::app);
```

[3] USE: Send/Receive data to/from files using the << and >> operators...

```
int one_number, another_number;
in_stream >> one_number >> another_number;
out_stream << "one_number = " << one_number;
```

[4] CLOSE: close every stream...

```
in_stream.close();
out_stream.close();
```

File Names as Input

Use a character array / C-String (array that ends in in the null character "\0")

[1] DECLARATION: char file_name[16]

<u>Argument</u>: Use one more than the maximum number of characters (additional char is for the null char).

<u>Initialize</u>: Can initialize when declared.

```
char var1[20] = "Hi There"; // need not fill all
char var1[] = "Hi There";
```

[2] INPUT:
```
cout << "Enter file name (15 char max): ";
cin >> file_name[16];
```

[3] USE the string variable (character array) as the argument to open:

```
in_stream.open(file_name);
```

Check File Open

Always follow a call to "open" with a test to assure it was successful.

- Use **.fail()** to test if the stream operation failed.
- Example:

```
in_stream.open("stuff.data");
if ( in_stream.fail() )
{
    cout << "Fail, maggot!";
    exit(1);
}
```

exit(int): exits program immediately
- *int* can be any integer; non-zero for errors

Member Functions that check the state of a stream:

.bad() Check whether badbit is set
.good() Check whether state of stream is good
.fail() Check whether either failbit or badbit is set
.eof() Check whether eofbit is set

Input Until EOF

Every input-file stream has an .eof() member function

EXAMPLE:

```
in_stream.get(var_next)
while ( ! in_stream.eof() )  // boolean
{
    // do something with each variable
    in_stream.get(var_next)
}
```

OR: Read numbers from a file until there are no more numbers to read (control stmt is input stmt, too):

```
while (in_stream >> var_next)
{
    // do something with each variable
}
```

cin.eof()
- EOF for cin: "**ctrl-z**" (Windows) or "**ctrl-d**" (Linux)
- Example: loop input until EOF

```
while ( !cin.eof() )
{
    cin >> str;
    ... do fancy things...
}
```

cin.ignore(int, char)

Used to ignore up to int characters or until char dilemeter.

Example: read in a fraction...

```
int numerator, denominator;
cin >> numerator;
cin.ignore(256, '/');
cin >> denominator;
```

That will read in an int, ignore up to 256 characters or until the '/' and then read in the next in.

Formatting Output

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

```
out_stream.setf(ios::fixed);
out_stream.setf(ios::showpoint);
out_stream.precision(2);
```

Change the precision by issuing a new .precision(4) statement. Don't need to set the flags again.

Formatting Flags

ios::fixed	floating-point numbers not written in e-notation
ios::scientific	floating-point numbers written in e-notation
ios::showpoint	always show decimal point and trailing zeros
ios::showpos	always show + for positive numbers
ios::right	right justified in space specified [this is DEFAULT behavior]
ios::left	left justified in space specified

Set Width

width - member function, must be accompanied by dot (.) operator; applies ONLY to the NEXT item output.

```
cout.width(4);
cout << 7 << endl;  // prints "   7"
```

setw() - manipulator (see below)

Manipulators

A manipulator is a function that is called in a non-traditional way and placed after insertion operator <<

endl
- Inserts a new-line character and flushes the stream

setw (int n);
- Sets field width to be used on output operations.
- n = number of characters to be used as field width.
- declared in header <*iomanip*>
- Example: cout << setw(4) << 10 << setw(4) << 30 ;

setprecision (int n);
- Sets the decimal precision used to format output floating-point values
- n = new value for the decimal precision.
- declared in header <iomanip>.
- Example:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout << "$" << setprecision(2) << 10.3 << endl
```

Arrays – General

Declare:
```
double score[5]  // Declare array of 5 integeres
```
DO NOT use variables in array declarations
```
cin >> number;
char score[number]; // use a dynam array (pointer)
```
You can, however, use CONSTANTS in array declarations
```
const int NUMBER = 5;
int score[NUMBER];
```
Initialize: ...when you declare it by using **{ }**
```
int children[3] = {3, 12, 1}  // can omit argument
int children[] = {3, 12, 1}   // same as above
```
Any array values not initialized are set to 0 of the base type.

Reference: `score[0]  // Ref to the int at first index`
- An indexed variable can be used any place an ordintary variable of this type can be used.
- Expressions can be used in the brackets as long as they evaluage to integers within the range of the array's indexes (score[n+1]).

Partially-Filled Arrays

Requires an additional call-by-reference value.
- Additional parameter in functions to keep track of how many elements of the array have been used.

Example:
```
void funct(int a[], int size, int& number_used);
// array "a" is passed with its declared size
// number_used is computed in the funcion
```

Multidimensional Arrays: "Pages"

1. Declaration: `char page[30][100]`
```
// declares 30 arrays of 100 elements each
```
2. Indexed Variable (will have two indexes)
```
SYNTAX:  page[which array][which element]
         page[row][column]
EXAMPLE: page[2][50]
         page[0][0], page [0][1], page [0][2] ...
         page[1][0], page [1][1], page [1][2] ...
         page[2][0], page [2][1], page [2][2] ...
```
3. As a function parameter:
```
void funct(const char p[][100], int size_dim_1);
// The size_dim_1 is the first dimension (not
   given in the first parameter when declared).
```

Arrays in Functions

You can pass entire arrays or indexed variables...

Indexed Variables:
```
my_funct(score[i]) // pass an indexed var to a funct
```
Entire Array:
The parameter is an ARRAY PARAMETER. See below...

Functions CANNOT Return Arrays
- You can return a pointer* to an array, though.

Arrays as Parameters (in Functions)

Indicated with empty square brackets. **[]**
- An array parameter functions similar to a call-by-reference.
- The array passed as an argument is changed by the function.

1. Function Declaration:
```
void fill_up(int a[], int size);
```
2. Function Call:
```
int score[5], array_size(5);
   // declares array & integer variables
fill_up(score, array_size);
   // array argument WITHOUT BRACKETS (see below*)
```
3. Function Definition:
```
void fill_up(int a[], int size) { ... }
```
* Just like the "&" for the call-by-reference is used only in the decl. and def., not in the call. The mechanism (array, call-by-ref) is established in the declaration and definition. The call just plugs a variable into that mechanism.

Constant Array Parameter:
- Used as a precaution to make sure the array is not changed when passed into the function
- Include the key word "const" in the Function Declaration/Definition:
```
void function(const int a[], int size_of_a);
```
[ ] is same as * in function parameters
```
void fill_up(int a[], int size);  // same
void fill_up(int *a, int size);   // same
```
- The array parameter is changed to a pointer by the compiler, so you can pass a pointer instead.
- An array variable is a pointer variable containing the address to the first element of the array.
- Example 1:
```
int a[10];
int *p;
p = a;  // p points to same address as a
assert( p[0] == a[0] ); // true
assert( p[1] == a[1] ); // true, etc.
```
- Example 2:
```
a = p; // illegal: cannot change array's pointer
```

Array of Pointers

```
int **array_ptr_var;
    // declare a pointer to pointers
array_ptr_var = new int*[100]
   // create 100 new pointers in the pointer array
```

Vectors: General

Vectors are like *arrays* that can *grow and shrink* while a program is running. A vector has a *base type* and stores values of that base type.

Library:
```
#include <vector>
using namespace std;
```
Declare: declare variable v, vector of type int...
```
vector<int> v;
```
Constructor: initialize 10 positions to 0...
```
vector<int> v(10);
```
Declare & Initialize:
```
vector<double> sample = {0.0, 1.1, 2.2};
```
Use:
```
v[i] = 42;  // square brackets: change elements
v[i];       // square brackets: access elements
```
Restriction:
You cannot initialize new elements in a current vector with the square bracket notation.
- You can only change elements that have already been assigned in the vector.
- To add new elements to the end of an existing vector use the member function **.push_back()**

Vector Member Functions

.push_back(value)
Adds elements to the end of a vector. Example:
```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```
.size() *sample.size()*
Returns number of elements in vector (returns unsigned int, not int; type cast if needed)

.capacity() *sample.capacity()*
Returns number of total potential elements (when size reaches capacity, capacity is doubled).

.reserve(num) *sample.reserve(num)*
Explicitly increase vector capacity to num

.resize(num) *sample.resize(num)*
Change vector size (up or down) to num.

Vectors & Loops

For-Loop:
```
for (unsigned int i = 0; i < sample.size(); i++)
    cout << sample[i] << endl;
```
Ranged For-Loop:
```
for (auto i : sample)
    cout << sample[i] << endl;
```

Vectors in Functions

Call-by-Value:
```
void func_name(vector<int> var);
```
Call-by-Reference:
```
void func_name(vector<int>& var);
```
Pointer:
```
void func_name(vector<int>* var);
```

C-Strings

A C-style string is simply an **array of characters** that uses a **null terminator** (‘\0’, ascii code 0).
- The null character serves as an end marker (sentinal value) and is placed immediately after the last char of the array (**not** the **final** char of the array)

H	o	w	d	y	**\0**				

- The null character distinguishes the C-string from a normal array of char.

C-String Variables

Declare:
```
char s[10]; // holds 9 letters + the null at the end
```

Declare & Initialize:
```
char msg[20] = "Howdy"; // partially filled C-String
                        // 20 char long including null
char msg[] = "Howdy";   // auto sizes the C-String to
                        // 6 (5 char + null)
```

Usage:
C-String variables can be partially filled or filled.

C-Strings: Assignment Operator (=)

You **cannot** use a C-string variable in an assignment statement (= only works when initializing).
- **C-Strings follow all the same rules as arrays.**

```
- Example:  char a_string[10];
            a_string = "Hello";  // ILLEGAL STMT. BAD!
```

C-String Input

cin
Will read input into a C-Str until the 1st whitespace.
```
char a[50], b[50];
cin >> a >> b;
```

.getline(cstring_var, max_char + 1, 'delim')

Reads entire line of input into a C-String variable.
- Reads in until 'delim" char or end of max_char in 2nd parameter (+1 for null char at end: '\0').
- Example:
```
char string[256];
cin.getline( string, 256, '\n' );
cout << string;
```

C-String Output

cout
Prints characters until it encounters the null char.

Example:
```
char name[20] = "Dipweed";
cout << "My name is: " << name << endl;
```

C-Strings: Predefined Functions

strcpy(destination, source);

Copy string: copies source into destination, including the null character (and stopping at that point).

Parameters:
- **destination:** destination array (or pointer to said array) where the content is to be copied.
  - Remember: an array variable (or cstring variable) is a pointer variable containing the address of the first element of the array.
  - Therefore passing an array/cstring variable to strcpy is in effect passing a pointer variable.
- **source:** Cstring (or "string literal") to be copied.

Return Value: destination is returned.

Examples:
```
strcpy (str2,str1);
strcpy (str3,"copy successful")
```

strcat(destination, source);

Concatenate strings: Appends source to destination.
- The terminating null character in destination is overwritten by the first character of source
- A null-character is included at the end of the new string formed by the concatenation of both in destination.

Parameters:
- **destination:** Cstring large enough to contain result
- **source:** Cstring (or "string literal" to be appended.

Return Value: destination is returned.

Examples:
```
char str[80];
strcpy (str,"these ");
strcat (str,"strings ");
strcat (str,"are ");
strcat (str,"concatenated.");
```

strcmp(c-str1, c-str2);

Compare two cstrings (lexicographically: alpha-num)

Return Value:
- **0** the contents of both strings are equal
- **<0** c-str1 has a lower value than c-str2
- **>0** c-str1 has a greater value than c-str2

strlen(c-str);

Returns the length of the cstring c-str.
- Length is up to but not including the null char.

sizeof() vs. strlen()
1. sizeof() returns the size of the entire array.
2. strlen() returns the number of characters before the null terminator.

Example:
```
char name[20] = "Greg";
sizeof(name);          // returns 20
strlen(name);          // returns 4
```

C-Strings: Predefined Functions

strncpy(destination, source, num);

Copy string: Copies the first num characters of source to destination.
- Destination padded with zeros if longer than source.
- No null-character is implicitly appended at the end of destination if source is longer than num.

Parameters:
- **destination:** destination array (or pointer to said array) where the content is to be copied.
- **source:** Cstring (or "string literal") to be copied.
- **num:** Max number of chars to be copied from source.

Return Value: destination is returned.

Examples:
```
// copy to sized buffer (overflow safe):
strncpy ( str2, str1, sizeof(str2) );

// partial copy (only 5 chars):
strncpy ( str3, str2, 5 );
str3[5] = '\0';
```

strncat(destination, source, num);

Concatenate strings: Appends the first num characters of source to destination, plus a null-character.
- If the length of the C string in source is less than num, only the content up to the terminating null-character is copied.

Parameters:
- **destination:** Cstring large enough to contain result
- **source:** Cstring (or "string literal") to be appended
- **num:** Maximum number of characters to be appended.

Return Value: destination is returned.

Examples:
```
char str1[20];
char str2[20];
strcpy (str1,"To be ");
strcpy (str2,"or not to be");
strncat (str1, str2, 6); //out: To be or not
```

strncmp(c-str1, c-str2, num);

Compare two cstrings (lexicographically: alpha-num) up to num characters.

Return Value:
- **0** the contents of both strings are equal
- **<0** c-str1 has a lower value than c-str2
- **>0** c-str1 has a greater value than c-str2

C-Strings-to-Number Functions

Requires: #include <cstdlib>

atoi	alpha to integer Example: int x = atoi(c_str_var);
atol	alpha to long Example: long x = atol(c_str_var);
atof	alpha to float/double Example: double x = atof(c_str_var);

C-Strings as Function Args/Parameters

Declaration:	void funcName(char cstr[]); // no &
Call:	funcName(cstr_var_name);
Definition:	void funcName(char cstr[]) // no & { ... do something ... }

[ ] is same as * in function parameters

Remember: a cstring is an array with a null char terminator. Therefore...

- An array variable is a pointer variable containing the address of the first element of the array.
- Therefore, you can use either the array or the pointer operator in the function:
- **Example:**

```
Declaration:    void funcName(char * cstr);
Call:           funcName(cstr_var_name);
Definition:     void funcName(char * cstr)
                { ... do something ... }
```

Multi-Dimensional Character Arrays

These are arrays of C-Strings.

Example: Array of 4 C-Strings 8-char long

```
char gasgiants[4][8];
```

Example: Array of 3 C-Strings 6-char long

```
char stooges[3][6] = {"moe", "larry", "curly"};
```

Strings: Standard Class string

```
#include <string>     // library, include
using namespace std;  // std namespace
string phrase;        // declaration
string word("ants");  // declaration & initialization
string word = "ants"; // same as above
```

String Operators

=	<u>Assign</u> value to string variable (s1 = "Hello";)
+	<u>Concatenate</u> two strings
<<	<u>Insertion</u>: output string objects cout << s1;
>>	<u>Extraction</u>: stops at whitespace; use getline() for lines cin >> s1;
== **!=**	Comparisons for equality or inequality. - returns a Boolean value (true/false)
< <= **> >=**	Lexicographical Comparisons: Alphabetical order.

Predefined String Function: getline()

Syntax: *getline (inputStream, stringVar, 'delim')*

Example:

```
#include <string>
using namespace std;
string string1
getline(cin, string1);
```

Extracts characters from input stream cin and stores them in the string variable string1 until the delim char or the newline *('\n') character is found.*

Note: getline() removes the newline from the input buffer.

cin.ignore()

cin does *<u>not</u>* remove the newline from the input buffer.

- If you follow cin with another cin, cin will just skip the first whitespace (newline) and read the input until the next whitespace. All is good.

Problem: If you follow cin with a string getline(), you need to *<u>first</u>* clear the input buffer of the newline byusing cin.ignore().

Syntax 1: cin.ignore()

With no arguments, skips the next input character.

Syntax 2: cin.ignore(n, delim)

n => max number of characters to extra and ignore. The default is 1 (i.e., w/o the n parameter).

delim => stops extracting/ignoring characters at this char (delim char *<u>is</u>* extracted/ignored).

Example:

```
cin >> string_var; // newline left in input buffer
cin.ignore(1000, '\n') // removes newline
getline(cin, string_var2) // line into variable
```

String Member Functions

STRING CONSTRUCTORS:

string str;	Declares empty str variable
string str("sample");	Declares and initializes str variable
string str(a_string);	Creates str that is a copy of a_string

STRING ACCESSORS:

str[i]	returns ref to char at index i
str.at(i)	returns ref to char at index i, checks for illegal index
str.substr(pos, len)	returns substring len long in chars from pos (w/o len: to end)
str.length()	returns the length of a string

STRING ASSIGNMENT/MODIFIERS:

str.empty()	<u>Boolean</u>: returns true if str is empty, false if not
str.insert(pos, str2)	insert str2 in str beginning at pos
str.erase(pos, len)	remove substring of size len beginning at pos
str.c_str()	returns corresponding C-string

STRING FINDS:

str.find(str1)	returns index of 1st* occurrence of str1 in str

* If not found, returns "string::npos". Usage Example: while(str.find(".")) == string::npos

str.find(str1, pos)	...same but search starts at pos
str.find_first_of(s, p)	returns index of 1st occurrence in str of any char in s, starting at position p
str.find_first_not_of(s, p)	returns index of 1st occurrence in str of any char not in s, starting at position p

String-to-Number Functions:

stoi(str)	string to integer <u>Example</u>: int x = stoi(c_str_var);
stol(str)	string to long <u>Example</u>: long x = stol(c_str_var);
stof(str)	string to float <u>Example</u>: float x = stof(c_str_var);
stod(str)	string to double <u>Example</u>: double x = stof(c_str_var);

References - &

What is a reference? A reference is an alias, or an alternate name, to an existing variable.

- **Objects:** References must always alias objects.
- **Main use:** as function parameters to support pass-by-reference.

Pointers & References

C++ inherited pointers from C.

- **When to use:** Use references when you can. Use pointers when you have to.
- **References:** preferred in a class's public interface
- **Pointers:** preferred in a class's private interface.

Pointers

A pointer is the memory address of a variable.

- The address points to the variable b/c it identifies the variable by telling WHERE the variable is, rather than what the variable name is.
- Call-by-ref arguments in functions are pointers.

Pointer Variables

- A pointer can be stored in a variable.
- The pointer itself is a memory address.

Pointers: Uses of the Asterisk (*)

1. In a declaration, the * defines a pointer:
```
double *p;
```
2. In an executable, the * dereferences a pointer
```
*p = 9.99;
```
3. In a function declaration or definition, the * can define a pointer return type:
```
type_name* func_name(type para1, type para2);
```
Note: The function will return a point of *type_name*

Declaring Pointer Variables

For a variable to hold a pointer, it must be declared a pointer type.

- Each variable type requires a different pointer type
- Pointer variables are declared like ord. variables but with an asterisk (*) in front of the variable.

Examples:
```
double *p;   // declares variable p of pointer type
int *p1, p2; // declares 1 pointer variable & 1 ord.
```

Dereferencing Operator: *

The * in front of a pointer variable produces the variable (the data) the pointer points to.

Example:
```
int *p, v(0);
p = &v; // v and p refer to same variable
*p = 42;
cout << v << endl;  // output: 42
cout << *p << endl; // output: 42
```

Copy Pointer Variables

```
int *p1, *p2, v(0);
p1 = &v;
p2 = p1;  // copies the address from p1 to p2
```

Address-Of Operator: &

The operator & in front of an ordinary variable produces the address of that variable (i.e., the pointer that points to that variable).

Example:
```
double *p, v;
p = &v;       // p now points to variable v
*p = 9.99     // sets the value of v to 9.99
```

Pointers in Functions

Pass a pointer to a function:

1. Declare the function ***parameter** as a pointer type.
2. Pass in the **&**address of the argument variable.
3. ***Dereference** in the definition to assign data.

Example:

1. Declaration: declare the parameter as a pointer
```
void func(int *param);
```
2. Call: pass in the address of the argument
```
func( &variable );
```
3. Definition: dereference pointer to assign data
```
void func(int *param)
*param = 10;
```

Pass-by-Reference with Pointer Arguments

A pointer is just a variable that holds an address.

- You would want to pass a pointer by reference if you need to modify the pointer rather than the object the pointer is pointing to.

Example:
```
void func(int*& param)
// param is a reference to a pointer of type int
```

Example (same as above but with typedef)
```
typedef int* IntPtr      // define pointer type
IntPtr p1;               // p1 is pointer type int
void func(IntPtr& param) // call-by-ref pointer
```

typedef

Define your own pointer type so you don't have to declare with an *.

- **typedef** can be used to define an alias for any type.

Example:
```
typedef int* IntPtr; // defines pointer type
IntPtr p;            // same as: int *p
```

Function: `void func_name(IntPtr& pointer_variable);`

Function Parameters: char* vs const char*

char * name

- A pointer that be changed and that also allows writing through it when dereferenced via * or [].
- You can change the char to which name points, and also the pointer value (address).

const char * name | char const * name

- A constant char pointer, or a char pointer that cannot be modified,
- The pointer can be changed but it does NOT allow writing through it when dereferenced via * or [].
- You can change the pointer, but not the char to which name points to.

const char * const name

- A constant pointer to a constant char (so nothing about it can be changed).

Dynamic Variables

Use: Generally used when the size of a variable/array is not known at compile time (only at run time).

- Dynamic memory allocation allows running programs to request memory from the heap (the large pool of unused memory allocated by the operating system).

Single Dynamic Variables

[1] new int;
- This dynamically allocates an integer (and discards the result).
- It requests an integer's worth of memory from the operating system.
- The **new** operator **returns a pointer** containing the address of the memory that has been allocated.
- We can assign the return value to a pointer variable to access the allocated memory later.

[2] int *ptr = new int;
- Dynamically allocate an integer's worth of memory.
- Assign the address returned by new to the variable ptr so we can access it later.

[3] *ptr = 7;
- Assign value of 7 to allocated memory.
- Dereference the pointer (with *) to access the memory.

Pointers: One of the main uses of pointer variables is to store the addresses of dynamic variables.

Initializing Dynamical Variables

Initialize a dynamic variable **when you allocate** it (using either direct or uniform initialization).

Example:
```
int *ptr1 = new int (5);   // direct initialization
int *ptr2 = new int { 6 }; // uniform initialization
```

Dynamic Variable: How To

```
  1. Declare:     int *p1;       // or use a typedef
  2. Initialize:  p1 = new int;
  3. Use:         *p1 = 42;      // sets value to 42
or. All in one:   int *p1 = new int { 42 }
  4. Delete:      delete p1;
  5. set nullptr  p1 = nullptr;
```

Deleting Dynamic Variables

When finished with a dynamically allocated variable (a variable created with **new**), you must explicitly tell C++ to free (deallocate) the memory for reuse.

For single variables, this is done via the scalar (non-array) form of the **delete** operator:
```
int *ptr = new int(1);
delete ptr;     // return the memory to O/S
ptr = nullptr;  // set ptr to be a null pointer
```

delete Operator

delete returns the memory being pointed to back to the operating system.
- The O/S is then free to reassign that memory.
- Note: delete does NOT delete the variable; it deallocates previously allocated memory.

Example 1:
```
int *ptr = new int; // dynamically allocate an int
*ptr = 7;           // put value in that memory
delete ptr;         // return the memory to O/S
                    // ptr is now a dangling pointer
cout << *ptr;       // BAD: Deref dangling pointer
delete ptr;         // BAD: deallocate memory again
```

Example 2:
```
int *ptr = new int;  // dynamically allocate an int
int *otherPtr = ptr; // otherPtr pointes same memory
delete ptr;          // return the memory to O/S
  // ptr and otherPtr are now dangling pointers
ptr = nullptr;
  // ptr is now a nullptr (no longer dangling)
```

Rule: delete + nullptr

After deleting the allocated memory, set all pointers that point to the deleted memory to nullptr.
- **nullptr: means** "no memory has been allocated to this pointer."
- if you use **delete**, follow it with **var = nullptr;**

Dynamic Array

Dynamically allocate arrays of variables.

Dynamically allocating an array allows you to choose the array length at run time (vs. compile time as with fixed arrays).

new[] & delete[]

To allocate an array dynamically, we use the array form of **new** and **delete**: **new[]** and **delete[]**

Initializing Dynamic Arrays

Initialize to 0:
```
int *array = new int[length]();
```

Initialize using initializer lists:
```
int fixedArray[5] = { 9, 7, 5, 3, 1 };
  // initialize a fixed array in C++03
int fixedArray[5] { 9, 7, 5, 3, 1 };
  // initialize a fixed array in C++11 (no = )
int *array = new int[5] { 9, 7, 5, 3, 1 };
  // initialize a dynamic array in C++11 (no = )
```

Caveat: in C++11 you cannot initialize a dynamically allocated char array from a C-style string:
```
char *array = new char[14] { "Hello, world!" };
```
This does not work in C++11 but works in C++14.

Dynamic Array Variable, Example

```
cout << "Enter a positive integer: ";
int length;
cin >> length;
int *array = new int[length]; // use array new[]
array[0] = 5;                 // set element 0 to 5
 // dynamic array functions similar to a fixed array
delete[] array;               // use array delete[]
                              // to deallocate array
array = nullptr;              // don't forget nullptr
```

Dynamic Array: How To w/ typedef

1. **Define:** a pointer type with typedef
2. **Declare:** a pointer variable w/the new pointer type
3. **Initialize:** call new and assign the dyanamic variable to your pointer
4. **Use** the pointer variable just like ordinary array variable
5. Call **delete[]**
6. Set **nullptr;**

Example:
```
typedef double* DoublePtr;  // Define custom type
DoublePtr p1, p2;           // Declare pointer vars
p1 = new double;        // creates dynamic variable
p2 = new double[10];    // creates dynamic array
...
delete p1;
p1 = nullptr;
delete[] p2;          // must include brackets!
p2 = nullptr;
```

Dyanamic Arrays in Functions

You do NOT need the & operator because the dynamic array is itself a pointer (like w/ fixed array).

Example:
```
     typedef double* dblPtr;
     double func_name(dblPtr doubleArray);
 or: double func_name(double doubleArray[]);
 or: double func_name(double *doubleArray);
```

Pointer Arithetic

An alternative way to manipulate dynamic arrays.

Example:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
for (int i = 0; i < array_size; i++)
    cout << *(d + i) << " ";
delete[] d;
```

This dereferences (*) each successive element in the dynamic array.

- The "+1" evaluates to the next memory address of the array "d".
- This only works with addition/subtraction (so "++" and "--" work).

Structures

```
STRUCTURE (variable declared from a STRUCT)
  |
  |-> DATA: Member Variables (properties)
```

Structure: A kind of simplied class. It is...
...like an object w/o member functions (data only)
...a collection of values of different types in 1 item
...similar to an associate array (key:value).

Structures: Definition

Define: use keyword ***struct*** + CamelCase for Struct_Tag.
- Usually defined outside all functions, including main (i.e., global).
- All member variables (properties) are **public** by **default**.

Example:

```
struct Structure_Tag
{
    type1 member_variable_name1
    type2 member_variable_name2
    ...
};  // do not forget semicolon!
```

The Semicolon: allows you to declare structure variables after the structure definition.

Example: The following creates a struct and declares two structures variables of this stuct type:

```
struct WeatherData
{
    double temp;
    double wind;
} data_point1, data_point2;
```

Structures: Declaration

Once a struct definition has been created, structures can be declared just like any other variable.

Syntax: `Structure_Tag struct_var1, struct_var2;`

Structures: Initialization

You can initialize a struc when you declare it (must define it first) or after:

1. Define:

```
struct Date
{
    int month;
    int day;
    int year;
};
```

2. Declare: `Date due_date;`

3. Initialize:

```
due_date.month = 12;
due_date.day = 31;
due_date.year = 2004;
```

OR

2,3. Declare & Initialize: curly braces & semicolon...

```
Date due_date = {12, 31, 2004};
```

Structures: Use

The member variables are specified and accessed via the dot operator.

Example: `structVar.member_variable = value;`

Structures: Arrays & Lists

Structures as array elements: The member variable is specified with the dot operator after the index.

Example:

```
structArrayVar[0].member_variable;
structArrayVar[1].member_variable;
```

Structures: In Functions

1. Structs as **Call-by-Value** parameters:

```
return_type func_name(struct_name param_name);
int get_interest(CD_account& the_account);
```

2. Structs as **Call-by-Reference** parameters:

```
void func_name(struct_name& param_name);
void get_data(CD_account& the_account);
```

3. Structs as **function return type** (must return complete struct):

```
CD_account build_struct( double balance, double rate,
double term)
{
    CD_account temp; // declare empty struct temp
    temp.balance = balance;     // build...
    temp.int_rate = rate;       // complete...
    temp.term = term;           // struct...
    return temp;     // return complete struct
}
```

Classes

```
OBJECT (variable declared from a CLASS)
   |
   |-> DATA: Member Variables (properties)
   |
   |-> FUNCTIONS: Member Functions (methods)
```

Class: A class is a type (like int or double) that you define and whose variables are objects.
- Classes are a form of "encapsulation" (combining various items--variables, functions, etc.--into a single package, like an object).
- Encapsulation: information hiding / abstraction.

Object: Objects are variables that have...

```
 1. Data      - called Properties  (member variables)
 2. Functions - called Methods     (member functions)
```

Library: Classes are types you define and they should behave much like the predefined types like int, double, etc.
- You can build your own personal library of class type definitions.
- Then you can use your personal types just like predefined types.

Classes: Operators

::	Scope Resolution Operator, used with a CLASS NAME.
.	Dot Operator, used with OBJECTS (class variables).
=	Assignment Operator, can assign objects or structures to other objects or structures. - It copies all member values.

Classes: Definition

To define a class, use key word ***class*** + CamelCase for class type name.
- Usually defined outside all functions, including main (i.e., global).
- Member functions are defined before member variables

Example:

```
    class Object_Name     // creates type Object_Name
    {
        public:
        type member_function_name(); // method decl.
        type member_variable_name;   // property decl.
    };  // do not forget semicolon!
```

Classes: Declaration

Once a class defintion has been defined, objects can be declared just like any other variable.

Syntax: `Object_Name object_var1, object_var1;`

Classes: Initialization

You assign values to member variables (properties) with the dot operator.

1. Define:

```
class Day_of_Year
{
    public:
    void output();
    int month;
    int day;
};
```

2. Declare: `Day_of_Year today, birthday;`

3. Initialize:

```
birthday.month = 9;
birthday.day = 17;
birthday.output();
```

Classes: Public vs. Private

Goal: Build enough member functions so users never need to access member variables directly (they will access them through member functions).

public: keyword making members (variables & functions) available to all users through the dot operator.

private: keyword limiting direct access to members (variables & functions).
- Member variables & functions are **private** by **default**
- Properties: all member variables should be private (restricted) & accessed via public member functions.
- Methods: private member functions can be used in any other member functions but nowhere else (restricted)

Example: All member variables are private and accessed or changed through public member functions.

```
  class Day_of_Year
  {
      public:
      void input();
      void output(ostream& out_stream); // uses <iostream>
      void set(int new_month, int new_day);
      int get_month;
      int get_day;

      private:
      void check_date();
      int month;
      int day;
  };

  void Day_of_Year::set(int new_month, int new_day)
  {
       month = new_month;
       day = new_day;
       check_date();
  }

  void Day_of_Year::output(ostream& out_stream)
  {
      // out_stream parameter can be replaced with
      // either cout or file stream (see p. 574)
      out_stream.setf(ios::fixed);
      out_stream.setf(ios::showpoint);
      out_stream.precision(2);
      out_stream << month << day;
  }
```

Classes: Member Functions

Declaration: Member ***functions*** are declared in the class definition before the main() function.

Definition: Member ***functions*** are defined below the main() function and must be scoped.

Scope Resolution Operator: Definitions must be SCOPED with the :: scope resolution operator.

Syntax:

```
    Return_Type Object_Name::member_function_name()
    {
        ...do something...;
        // can use all members of class w/o dot oper.
    }
```

Example:

```
void Day_of_Year::output()
{
    cout << "Month: " << month
    << ", Day: " << day;
}
```

Classes: Calling Member Functions

When you call a member function you always specify the object.
- Use dot (.) operator to specify the calling object
- The calling object determines the meaning of the function name.

Syntax:

```
  Calling_Object.Member_Function_Name(Argument_List);
```

Example:

```
in_stream.open("infile.dat");
out_stream.open("outfile.dat");
out_stream.precision(2);
```

Classes: Constructors

Constructors are functions used to initialize some or all member variables when the object is declared.

A contructor is a member function defined within the class and automatically called when an object of that class is declared.

Classes: Constructors -- Declaration

Example:

```
  class BankAcc
  {
    public:
    BankAcc(int dollars, int cents, double rate);
    BankAcc(); // overloaded (default) constructor

    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();
    double get_balance();
    double get_rate();
    void output(ostream& outs);

    private:
    double balance;
    double interest_rate;
    double fraction(double percent);
  };
```

Classes: Constructors -- Definition

Constructors are defined like other member functions ***except***...

1. A constructor must have the ***SAME NAME*** as the class.
2. A constructor definition ***CANNOT RETURN A VALUE*** (no return type).

Example:

```
class BankAcc
{
    BankAcc(int dollars, int cents, double rate);
    BankAcc();
    ... etc ...
};
BankAcc::BankAcc(int dollars, int cents, double rate)
// no return type given for constructors
{
    if ((dollars < 0 || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values.\n";
        exit(1);
    }
    balance = dollars + ( 0.01 * cents );
    int_rate = rate;
 }
BankAcc::BankAcc() : balance(0), int_rate(0)
// initialization section: colon + comma-sep variables
 {
    // body intentionally left blank
 }
```

Classes: Constructors -- Initialization

Example: Calling the constructor...

```
 BankAcc account1(10, 50, 2.0);   // or...
 account2 = BankAcc(500, 0, 4.5); // same as above
  // account1 and account2 delcared
  // contructor is called
  // initial member variable values set to those given
```

Example: Calling the *__default__* constructor (no args)...

```
 BankAccount account1;     // no parens (like "int a")
 account2 = BankAccount(); // parens w/ explicit call
```

Classes: Copy Constructors

A constructor that has one parameter of the same type as the class.
- The one parameter must be call-by-reference
- Normally the parameter is preceded by const so as to avoid inadvertent modification.
- Whenever C++ needs to make a copy of an object, it calls the copy constructor.

Example, Prototype:

```
      ClassName(const ClassName& copiedObject);
```

Example, Definition:

```
      ClassName(const ClassName& copiedObject)
      {
          setVar1(copiedObject.var1);  // or...
          var1 = copiedObject.var1;    // etc.
      }
```

Example, Usage (declare & initialize new object):

```
      ClassName newObject(copiedObject); // new copy
```

Inheritance: Derived Classes

Inheritance allows you to define a general (parent) class and then define more specialized (child) classes that add details to the existing general (parent) class.

1. **General Class:** Parent or Base Class
2. **Derived Class:** Child Class (inherits the Parent's member functions)

DEFINITION of a derived class

- Syntax:

```
    class ChildClassName : public ParentClassName
    {
       public:
       // additional child properties and methods
       private:
       // additional child properties and methods
    };
```

All members of the ParentClass will be available to the ChildClass.

USE of a derived class

1. Declare and Initialize:

ChildClassName objectVariableName(val1, val2, val3);

2. Call member functions with ChildClassName (not parent)

objectVariableName.memeberFunction();

```
// from child or parent
// all parent member functions are available to the
   child via the dot operator w/the child class name
```

Limitation: Derived classes **cannot** access the **private** members of their base classes.
- No type of inheritance allows access to private members.
- However, a "**friend**" declaration allows this.

Destructors

A member function of a class that is called automatically when the class goes out of scope.
- Used to eliminate any dynamic variables (via call to delete) created by the object (return memory to heap).
- Can also perform other clean-up tasks (like closing i/o file streams).
- This function is never called by the programmer. It is always called by the compiler.

Syntax: same name as class preceded by the tilde (~) and followed by double parens().
- No type for a return value (just like constructors).
- No parameters.
- One destructor allowed per class.

Arrow Operator ->

Specifies a member of a struct (or a member of a class object) that is pointed to by a pointer variable.

Syntax: Pointer_Variable->Member_Name;

Example:

```
1. Define a structure
   struct Record
   {
       int number;
       char grade;
   }
2. Create a Dynamic Variable
   Record *p;
   p = new Record;
   p->number = 2001;
   p->grade = 'A';
```

this Pointer

The ***this*** pointer is a predefined pointer that points to the calling object.
- Used when defining member functions for a class and you want to refer to the calling object.

Overloading the = Operator

The overloaded assignment operator (=) must be a member of the class, not a friend.
- The right side of the = is the parameter/argument when called.
- The left side of the = is the calling object ("this" inside the function). This means the call works as if there were a dot operator (left_side.=(right_side)), but there is not (left_side = right_side).

Declaration (in class definition):

```
    ret_type operator =(const Class_Name& right_side);
```

Definition (in implementation file):

```
    ret_type Class_Name::operator =(const Class_Name&
    right_side);
```

Separate Compilation

Comprised of the use of three files:

```
1. Interface File        .h     (header)
2. Implementation File   .cpp
3. Application File      .cpp   (driver)
```

Separate Compilation: How-To

1. Interface File: **dtime.h**

- Include class definitions, function declarations & overloaded operators.
- Comments to explain how they all work
- Includes the #ifndef/#endif to define the header

```
#ifndef DTIME_H
#define DTIME_H
< ... class def, etc. ...>
#endif
```

2. Implementation File: **dtime.cpp**

- Include definitions of functions and overloaded operators.
- Must contain the header file include directive:

```
#include "dtime.h"
```

3. Application File: **timeDemo.cpp**

- Driver program that uses the classes, functions, and overloaded operators of above .h/.cpp combo.
- Must contain the header file include directive:

```
#include "dtime.h"
```

Separate Compilation: Header Directive

Include in your header file to avoid compiling multiple copies of your header file.

#ifnedf HEADERNAME_H

- "if not defined": place on first line of header
- Follow with HEADERNAME_H (header name in caps)

#define HEADERNAME_H

- Directive that "defines" (labels) HEADERNAME_H

#endif

- "end if": place on last line of header

Example:

```
#ifndef DTIME_H
#define DTIME_H
< ... class def, etc. ...>
#endif
```

Namespaces

A namespace is a collection of name definitions
- e.g., class definitions, variable declarations

Namespaces: Creating

Place your code in a namespace grouping above main() or in a header file. Example:

```
namespace Name_Space_Name
{
    Some_Code;
}
```

Some_Code is now in the namespace Name_Space_Name and can be made available through a namespace directive:

```
using namespace Name_Space_Name;
```

Namespaces: Qualifying Names

1. Using Directive

Makes ALL names in the namespace available. Placed at beginning of file or {code block}

```
using namespace std;
```

2. Using Declaration

Makes only ONE name in the namespace available. Placed at beginning of file or {code block}

```
using std::cout;
```

3. Specific Qualifier

Makes one name available for one specific instance. Place in a statement/executable (in a line of code).

```
std::istream
```

Namespaces: Unnamed Namespace

Defined just like a normal namespace but w/o a name:

```
namespace
{
    Some_Code;
} // unnamed namespace
```

All the names definied in the unnamed namespace are local to the compilation unit (all the files #included in compilation).

- Any name defined in the unnamed namespace can be used without qualification anywhere in the compilation unit.
- There is ONE unnamed namespace in each compilation unit.

Namespaces: Global Namespace

Names in the global namespace have a global scope (all the program file) and can be accessed w/o a qualifier.

Difference: Names in an unnamed namespace are local to a complication unit.

Conditional (Ternary) Operator

Syntax: (expression 1) ? expression 2 : expression 3

If expression 1 evaluates to true,
then expression 2 is evaluated.

If expression 1 evaluates to false,
then expression 3 is evaluated instead.

Example: pick which value to assign to a variable...

```
int foo = (bar > bash) ? bar : bash;
```

If bar is bigger, bar is assigned; else bash assigned.