C++ Beginning Basic Summary Sheet

Responsible Party: Greg Kedrovsky

C++ Program Structure

```
#include <directive> // brackets for prepkg modules
#include "directive" // dble-quotes for mods you write
using namespace std; // Savitch 228-29
// prefer namespace indicator std:: (Carrano p.747)

const variable DECLARATIONS;

struct Structure_Tag
{
    type member_name ...
};

function_declarations();

int main()
{
   ... // do something
   return 0;
}

function_definitions()
{  ...  }
```

Basics

Variable Declarations (which is also a Definition):

- Variables must be declared before use.
- Declaring the variable tells the compiler what *type* of data to store and assigns it memory (defines it).
- 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);
```

Named Constants: variables that cannot be changed

```
const type_name VARIABLE_NAME = constant_value;
const double PI(3.14159);  // or, same thing...
const double PI = 3.14159; // ... same as above
```

Raw String Literals:

- Convenient if you have multiple escape sequences in a statement.
- 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";)

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.

doxygen

Documentation scheme applied to formatted comments.

1. Comment must be placed before public class/methods.
2. Comment must be contained within: /** ... */

doxygen tags: (Carrano 809ff)

```
@author lists the name of the class's programmer
@file   IDs the name of the class file
@param  IDs method's parameter by name + description
@post   IDs a method's postcondition
@pre    IDs a method's precondition
@return describes the method's return value
@throw  lists an exception that a method can throw
```

Data Types

Commonly Used Types

int	integer
double	integer plus decimal
char	one character
bool	false (0) or true (non-zero)

Type Casting: Changing Data Types

```
static_cast<new_type>(expression);
new_double = static_cast<double>(var_of_type_int);
```

Derived Types

Use the keyword **typedef** to give another name to an existing (primitive) data type.

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

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

Function: void func_name(IntPtr& pointer_variable);

Enumerations

Enums are a way to name **integer constants** rather than by doing it one by one.

Syntax:

```
enum {SUN, MON, TUE, WED, THU, FRI, SAT};
```

is equivalent to:

```
const int SUN = 0;  // Note: enums start at 0 and
const int MON = 1;  // are consecutive by default
...
const int SAT = 6;
```

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)	

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.

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
```

Linux: Build Process

1. The **COMPILER** takes the source (.h and .cpp) files and outputs object files.
2. The **LINKER** takes the object (.o) files and creates an executable file.

Linux: Simple Compilation with g++

Single File C++ Programs

Format1: g++ filename.cpp // default output: a.out
Execute: ./a.out

Format2: g++ filename.cpp -o outputFilename
Execute: ./outputFilename

Linux: Compilation with make

Allows you to only re-compile only the elements of your program that were modified.

1. Run by itself, **make** looks for a **makefile** in the current directory.
2. Run with **-f** switch, **make** looks for a specific makefile. Example: make -f Makefile_01

Linux: makefile

The **make** command will always run the commands from the first target in your **makefile**. This is how you build dependencies into the **makefile**.

makefile Syntax:

```
target: dependencies
[tab] system command
```

Note: You must offset the system command with a tab, not with spaces. (vim: **Ctrl-v + <tab>**)

makefile Example:

```
all: runMe

runMe: account.o driver.o
    g++ account.o driver.o -o runMe

account.o: account.cpp account.h transaction.h
    g++ -c account.cpp

driver.o: driver.cpp
    g++ -c driver.cpp

clean:
    rm *o runMe
```

makefile Explanation:

1. Running **make** in this **makefile**'s subdirectory runs **all** and seeks to fulfill the **runMe** dependency.
2. **runMe** depends on **account.o** and **driver.o** so they must be linked (g++ account.o driver.o).
3. In order to link **account.o** and **driver.o**, those two object files must be compiled (g++ -c).
   - If those object files already exist, they will not be compiled again unless their source (.h, .cpp) files have been modified.
4. Running **make clean** will delete your object files and the executable.

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 (namespace indicator: std::)
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.

Input: >>

input operator >>

```
cin >> x;  // read val for x from std input
cin >> x >> y: // read x val then y val
```

whitespace: input operator >> skips whitespace

boolean: cin >> x

Input operator returns true if successful, else false.

INPUT Member Functions

.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 whitespace: spaces and newlines (\n).
- .get() takes one argument: char variable to receive the character input from stream. **Example:**

```
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' );
```

Used to keep console window open at end of program:

```
std::cin.get() // forces key entry to proceed
```

cin.ignore(int, char)

Ignore up to int characters or until char delimiter.

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.

.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').

I/O Member Functions

All data is input and output as CHARACTER data.

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

cin

cin.clear()	clears error flag on cin
cin.ignore (1000, '\n')	clear eol in cin buffer

cin.ignore()

cin does *not* 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 *first* clear the input buffer of the newline by using 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 extract and ignore. The default is 1 (i.e., w/o the n parameter).

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

Example:

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

Output: <<

output operator <<

```
cout << "Prints to stdout " << ch1 << endl;
```

OUTPUT Member Functions

.put()

Outputs one character to the indicated output stream

- Takes one argument that should be a char expression
- Examples:

```
char ch = 'a';
cout.put(ch);  // displays a
cout.put('b'); // displays b
```

.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);
```

Manipulators

Used to gain more control over the format of output.

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

Syntax: std::cout << std::*manipulator*;

Example: std::cout << std::showpoint;

Most are declared in header <*iomanip*>

endl	insert new line, flush stream
fixed	fixed decimal point for float output
left	left-align output
right	right-align output
scientific	exponential (e) notation for floats
setfill(f)	set fill character to f
setprecision(n)	floating-point precision set to n
setw(n)	set field width to n Ex: cout << setw(4) << 10;
showpoint	show decimal point in float output
showpos	show + with positive integers
ws	extract whitespace char (input only)

Except for setprecision(n), manipulators only affect the appearance of the next set after the << .

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

Formatting Output: Example

```
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.

File I/O Includes

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

OR [1] & [2] DECLARE & CONNECT:

```
ifstream in_stream("infile.dat");
ofstream out_stream("outfile.dat");
```

[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 Window

Once a text file is open, the "**file window**" is positioned over the first character.

- The "**file window**" for a text file moves from character to character (always over the character you will read in next).

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]

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

Initialize: 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;
```

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

```
ifstream in_stream(file_name);
```

Check File Open

Always test to make sure your file opened as intended.

Method 1:

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

Method 2: Use **.fail()** to test if the stream operation.

```
ifstream in_stream("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

Character Input

Open a file: ifstream in_stream(file_name);

Read character by character:

```
char ch;
in_stream >> ch;   // preferred method
in_stream.get(ch); // also works
```

Explanation:

1. ch = value at the file window
2. Advances the file window to next character

Input Method: .peek()

Returns the character in the file window but does NOT advance the window to the next character.

Input until EOLn:

```
while ( in_stream.peek() != '\n' )
  cout << ( in_stream.get() ); // get char & display
```

Input until EOF:

```
while ( in_stream.peek() != EOF )
  cout << ( in_stream.get() ); // get char & display
```

Input Method: .ignore()

Advances the file window one character, skipping characters until it reaches the specified character.

1. **ignore()**: skips characters until end of line.
2. **ignore(n)**: skips characters until nth character OR end of line.
3. **ignore(n, 'c')**: skips characters until nth character, the character 'c', or end of line (whichever is first).

Example:

```
in_stream.ignore(10, '\n'); // advance 10 or to EOLn
```

Input Method: String getline()

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

Example:

```
ifstream in_stream(file_name);
string string1
getline(in_stream, string1);
```

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

Input Method: C-String .getline()

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

Example:

```
ifstream in_stream(file_name);
char cstr[256];
in_stream.getline( cstr, 256, '\n' );
cout << string;
```

Reads entire line of input into a C-String variable.
- Reads input until 'delim" char or end of max_char in 2nd parameter (+1 for null char at end: '\0').

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 input by user...

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

Character Ouput

Open new file: ofstream out_stream(fileName);
- File window is placed at beginning of new empty file

Append: ofstream out_stream(fileName, ios::app);
- File window is placed at end of the existing file.

Write character by character:

```
out_stream << ch;   // preferred method
out_stream.put(ch); // also works
```

Explanation:

1. Write the value of ch at the file window location
2. Advances the file window to one character

SELECTION

if ... else if ... else

If... Else If... Else...

```
if (bool exp)                (parenthesis) are required
{                            {braces} if compound stmt
   ...do something;
}
else if (bool exp)
{
   ...do something else;
}
else
{
   ...do something else by default;
}
```

switch

Usage:

- Choosing among more than two courses of action.
- Choice is to be made according to the value of an integral expression (bool, char, int, enum)
- Leave out "break" to apply one statement sequence to more than one case.

Syntax:

```
switch (switch_expression)
{
    case integral_expression_1:
        statement_sequence_1;
    case integral_expression_2:
        statement_sequence_2;
        break;
    ...
    case integral_expression_n:
        statement_sequence_n;
        break;
    default:
        default_statement_sequence;
}
```

***switch_expression* can be:** bool, enum, int, char

ITERATION

while

Used when evaluation is needed prior to execution

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

do

Used when execution is needed prior to evalution

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

for

Counted Loops

Syntax: *for (initialize; bool_eval; update)*

Example:

```
for ( int n = 1; n <= 10; n++ )
{
    // do something in each iteration
}
```

Range-Based For Loop (simplifies 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
```

MISC

Break & Exit

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

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

FUNCTIONS: General

1. function declaration:

```
type_returned function_name(type para, type para);
// 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(type para, type para)
  {
     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 & Arguments

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

- Can be pass-by-value, pass-by-reference, or mixed.
- Pass-by-reference indicated by an "**&**".

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

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

- The function makes local copies of the values of the arguments.
- Thus the function cannot alter the argument's value
- Arguments can be variables or literals.

Pass-By-Reference: (*type**&** parameter*) | (int& num1)

- The function does not copie the value of the argument; it references the argument location.
- This allows the function to change the values of the arguments.
- Arguments must be variables (not literals).
- **Usage:**
  1. When you want to change the value of an argument
  2. When the argument is too big to copy (ex: class)
- **const**: Preface the parameter with **const** if you do not want the function to change a pass-by-reference

```
void f(const int& x, int y, int& z);
// x – pass-by-reference, cannot change
// y – pass-by-value, local copy made
// z – pass-by-reference, can be changed
```

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 **PASS-BY-REFERENCE**; they cannot be pass-by-value. Ex:

```
input( istream& ins );
output( ostream& outs );
```

Constant Calling Object: const

Place the keyword "**const**" at the **end** of the function declaration.

- **const** must be included in BOTH the function declaration and definition.
- This indicates to the compiler that **the function cannot change the calling object**.

Example:

```
void output(ostream& outs) const;      // declaration
void Money::output(ostream& outs) const // definition
{ ...output routine...}
```

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
```

Friend Functions

A construct used to give non-member functions the same access privileges as a member function (in a class).

- So they can, for example, access private member variables directly.
- You simply make the function a "friend" of the class you want to access.

Explanation:

1. They are NOT members of the class (they are ordinary functions).
2. But they have access to the private members of the class (access and change).

Friend Functions: Rules

1. Use a **member function** if the task performed involves **only ONE object**.
2. Use a **friend function** if the task performed involves **MORE THAN ONE object** (like comparing two objects).
3. Overloaded operators: When the **invoking object** (left operand) is **not a member** of the class, the function **must** be declared as a friend function.

Friend Functions: Declaration

Friend functions are declared as "friends" INSIDE the class definition (header.h file) using the **keyword "friend"** in front of the declaration.

- It can be placed in either the public or the private section.
- It will always be considered publicregardless.

Example:

```
class DayOfYear
{
  public
  friend bool equal(DayOfYear date1, DayOfYear date2);
  ...
  private
  int month, day;
}
```

Friend Functions: Definition

Friend functions are defined outside the class definition (implementation.cpp file; **NOT scoped to the class**) without the keyword "friend".

Example:

```
 bool equal(DayOfYear date1, DayOfYear date2)
 {
    return ( date1.month == date2.month &&
             date1.day   == date2.day );
 }
```

Friend Functions: Call (usage)

The friend function is called WITHOUT the dot operator (it is NOT a member function).

Example:

```
    if ( equal( myDate, yourDate ) )
    { ... do something... }
```

Friend Functions: Invoking Operators

Rule: When the invoking object (left operand) is *not* a member of the class, the function *must* be declared as a **friend function**. Examples...

operator notation	function notation
a + b	a.operator+(b)
a + 5	a.operator+(5)
5 + a	operator+(5,a)

In the last example, no member object calls the operator, therefore it must be a **friend function**.

Arrays – General

Bracket Notation: **[ ]** - indicates an array

Declare: must indicate data type and array size.

```
double score[5]  // Declare array of 5 integers
```

YOU CANNOT use variables in array declarations

```
cin >> number;
char score[number]; // illegal, will not compile
// if you must use a variable, use a dynamic array
```

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 ordinary variable of this type can be used.
- Expressions can be used in the brackets as long as they evaluate 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 function
```

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).
```

Array Elements 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

Indicate the parameter is an array with the bracket notation: **[ ]**

- An array parameter functions similar to a call-by-reference (b/c it's a pointer to the array).
- The array passed as an argument is changed by the function.
- You must also pass an int with the capacity or the size of the array.

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> sample1 = {0.0, 1.1, 2.2};
vector<int> sample2(5, -1); // 5 elements of -1
```

Use:
```
v[i] = 42;  // square brackets: change elements
v[i];       // square brackets: access elements
```

Restriction on bracket notation [ ] :
You cannot initialize new elements in a current vector with bracket notation.
- You can only change elements that have already been assigned in the vector and/or access elements.
- 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);
```

.pop_back(value)
Removes the last element and shortens the vector.
- Does not give you the removed elements! It's lost!
- If you want the removed element, get it before pop.
```
int length = sample.size();
double last = sample[ length – 1 ]; // get last
sample.pop_back(); // reduce vector by one elem.
```

.size() *sample.size()*
Returns number of elements in vector (***returns unsigned int***, not int; static cast if needed; p. 720)
static_cast<int>(variable);

.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.

.clear() *sample.clear()*
Remove all vector elements and leave vector empty.

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);
```

Strings: Standard Class string

```
#include <string>     // library, include
using namespace std;  // std namespace or use std::
string phrase;        // declaration
phrase = "a phrase";  // initialization / assignment
string word("ants");  // declaration & initialization
string word = "ants"; // declaration & initialization
```

String Operators

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

String 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 ***not*** 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 ***first*** clear the input buffer of the newline by using 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 extract and ignore. The default is 1 (i.e., w/o the n parameter).

delim => stops extracting/ignoring characters at this char (delim char ***is*** 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 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
str.size()	same as str.length(): length/size

String Assignment / Modifiers

str.empty()	Boolean: 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

stoi(str)	string to integer Example: int x = stoi(c_str_var);
stol(str)	string to long Example: long x = stol(c_str_var);
stof(str)	string to float Example: float x = stof(c_str_var);
stod(str)	string to double Example: double x = stof(c_str_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 (sentinel 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 type 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 input 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: strlen

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 (length of string).

Example:

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

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"};
```

C-Strings: strcpy

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: char str1[]=”Whatever”, str2[20], str3[20];
          strcpy(str2,str1);
          strcpy(str3,"copy successful")
```

C-Strings: strncpy

strncpy(destination, source, num);

Copy string: Copies the first num characters of source to destination.
- Destination padded with zeros if longer than source.
- Beware (!) that if you hit the limit num before you get a NUL terminator on the source string, your destination string will NOT be NUL-terminated.

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.

Example: copy to sized buffer (overflow safe):

```
strncpy( str2, str1, sizeof(str2) );
```

Example: partial copy (only 5 chars):

```
strncpy( str3, str2, 5 );
str3[5] = '\0';
```

C-Strings: strcat

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.");
```

C-Strings: strncat

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
```

C-Strings: strcmp

strcmp(c-str1, c-str2);

Compare two cstrings (lexicographically: alpha-num)

Return Val:
- **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: strncmp

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

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.

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 - *

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, however, is a memory address.

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 (*) between type and 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
```

Pointers: Uses of the Asterisk (*)

1. In a declaration, the * **defines** a pointer:
   `double* p; // reads: declare a double pointer 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 pointer of *type_name*

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
```

The use of the **&** as the "address of" operator is an ***unsafe practice***.

- It results in a pointer to a local variable in stack memory.
- Best practice: Use pointers to point to variables in the free store only (variables created with **new**).

Copy Pointer Variables

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

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);
void func(int* param); // same as above
```

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)
void func(int *&param) // same as above
// 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
```

Pointer Arithmetic

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).

Function Parameters: char* vs const char*

char* name

- A pointer that be can 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).

Review of these constructs will lead to alcoholism.

Arrow Operator ->

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

- It is shorthand for:
  1. deference the pointer (to get the object), then
  2. use the dot operator to access object members
- Because of the order of operations, the deference must be in parentheses to be executed first.

Syntax: `Pointer_Variable->Member_Name;`

Example:

1. Define a Structure

```
struct Record
{
    int number;
    char grade;
}
```

2. Create a Dynamic Variable

```
Record *p;       // ptr var of type Record
p = new Record;  // ptr assigned to var p
p->number = 20;  // assign value to prop
(*p).number = 20 // same as above
p->grade = 'A';  // assign value to prop
(*p).grade = 'A'; // same as above
```

Dynamic Memory Allocation

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).

Dynamic Variables: Quickie Overview

```
1. Declare:     int* p1;      // or use a typedef
2. Initialize:  p1 = new int; // returns address
3. Use:         *p1 = 42;     // sets value to 42
or. All in one: int* p1 = new int(42);
4. Delete:      delete p1;    // frees heap memory
5. set nullptr  p1 = nullptr; // kind of placeholder
```

Dynamic Variables: Usage

[1] new int;

- This dynamically allocates an integer (and discards the result).
- It requests an integer's worth of heap memory from the operating system.
- The **new** operator **returns a pointer** containing the address of the heap memory the O/S 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.

Dynamic Variables: Initialize

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 Variables: Delete

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

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
```

new Operator

Example: new int;

- The **new** operator **returns a pointer** containing the address of the heap memory the O/S allocated.

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

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: BAD stuff...

```
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: GOOD stuff...

```
int* ptr = new int;  // dynamically allocate an int
int* otherPtr = ptr; // otherPtr points to same mem
delete ptr;          // return the memory to O/S
                     // both are now dangling ptrs
ptr = nullptr;       // ptr is now a nullptr (no
                     //  longer dangling)
```

Dangling Pointers:

Pointers that point to an address in memory that the O/S has not allocated (given to) the pointer.

- The O/S could have allocated (given) the memory address to another program.
- Therefore, dereferencing the pointer can give unexpected results.

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[] Operators

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

new[] **delete[]**

Dynamic Arrays: Initialize

Initialize to 0: (allocates memory for an empty array)

```
int* array = new int[length];
```

Initialize using initializer lists:

Example 1: initialize a fixed array in C++03

```
int fixedArray[5] = { 9, 7, 5, 3, 1 };
```

Example 2: initialize a fixed array in C++11 (no =)

```
int fixedArray[5] { 9, 7, 5, 3, 1 };
```

Example 3: initialize a dynamic array in C++11 (no =)

```
int* array = new int[5] { 9, 7, 5, 3, 1 };
```

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: Example

```
cout << "Enter a positive integer: ";
int length;
cin >> length;
int *array = new int[length]; // use new[] operator
                              // to allocate memory
array[0] = 5;                 // set element 0 to 5**
delete[] array;               // use delete[] operat.
                              // to deallocate memor.
                              // ptr is now dangling
array = nullptr;              // don't forget nullptr
```

**** dynamic arrays function similar to a fixed arrays**

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 dynamic 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);
```

Structures

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

Structure: A kind of simplified class. It is...
...like an class 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** (b/c there are no methods to access them).

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 struct type:

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

Structures: Declaration

Once a struct definition has been created (defined), structures can be declared 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 (local copy):

   sytax:
   return_type func_name(struct_name param_name);

   example:
   int get_interest(CD_account the_account);

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

   sytax:
   void func_name(struct_name &param_name);

   example:
   void get_data(CD_account &the_account);

3. Structs as **function return type** (must return a 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: 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 definition 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: Members, 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 member variables (properties) and member functions (methods).
- Class properties and methods are **private** by **default**.
- Properties: all member variables should be **private** (restricted) & accessed via public member functions.
  Note: private member variables may be used in the definition of ***any member function***, but nowhere else.
- 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 (Savitch p574)
    out_stream.setf(ios::fixed);
    out_stream.setf(ios::showpoint);
    out_stream.precision(2);
    out_stream << month << day;
}
```

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. - PROBLEM: If a member variable is a POINTER, you need to overload the = operator accordingly.

Classes: Member Function - Declaration

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

Classes: Member 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: Member Function - Call

When you call a member function you always specify the object with the dot (.) operator.
- 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: Static Member Variables

Member variables of a class can be made static by using the **static** keyword.

- Static member variables are shared by all objects of the class.
- Static members exist even if no objects of the class have been instantiated. They are created when the program starts, and destroyed when the program ends.
- Think of static members as belonging to the class itself, not to the objects of the class.

Public vs. Private

1. **Public:** If the static member variables are public, we can access them directly using the class name and the scope resolution operator.
2. **Private:** If the static member variables are private, a static member function is necessary.

Classes: Static Member Functions

Like static member variables, static member functions are not attached to any particular object.

- Static member functions can be **called** directly by using the class name and the **scope resolution operator**.
- Like static member variables, they can also be called through objects of the class type, though this is not recommended.

Two Quirks:

1. Because static member functions are not attached to an object, they have **no "this" pointer** ("this" points to the calling object and static members have no calling object; they belong to the class itself).
2. Static member functions can only access static member variables (b/c non-static member variables must belong to a class object, and static member functions have no class object to work with).

Static Example: Ssn Class SEPARATOR

HEADER FILE (Ssn.h): static member declaration

```
class Ssn
{
  public:
    static char getSEPARATOR(void);
    static const char PUBLIC_SEP;
    ...
  private:
    static const char SEPARATOR;
     // use keyword static in the declaration
     // whether declaring public or private
    ...
}
```

IMPLEMENTATION FILE (Ssn.cpp): define & initialize

```
const char Ssn::SEPARATOR = '-';  // member variables
const char Ssn::PUBLIC_SEP = '+'; // NO keyword static
char Ssn::getSEPARATOR(void)      // member functions
{
  return SEPARATOR;
  // static functions CANNOT access member data
  // static functions ONLY access static data
}
```

DRIVER FILE (driver.cpp): usage

```
  cout << Ssn::PUBLIC_SEP
    // no object needed, just call with scope
  cout << Ssn::getSEPARATOR();
   // static functions called with scoping (preferred)
  cout << a.getSEPARATOR();
   // static functions called using an object (bad)
```

Classes: Constructors

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

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

Constructors are defined like any other method ***except***:

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

Classes: 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().

```
~BankAcc();
```

- No type for a return value (just like constructors).
- No parameters.
- One destructor allowed per class.

Classes: Constructors -- Declaration

Example:

```
class BankAcc
{
  public:
    // Default Constructor
    BankAcc();
    // Argument Constructor
    BankAcc(int dollars, int cents, double rate);

    // Destructor
    ~BankAcc();

    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

Example, Default Constructor:

```
BankAcc::BankAcc() : balance(0), int_rate(0)
// initialization section: colon + comma-sep variables
 {
    // body intentionally left blank
 }
```

Example, Argument Constructor:

```
BankAcc::BankAcc(int dollars, int cents, double rate)
{
    if ((dollars < 0 || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values.\n";
        exit(1);
    }
    balance = dollars + ( 0.01 * cents );
    int_rate = rate;
 }
```

Classes: Constructors -- Initialization

Example: Calling the default constructor (no args)...

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

Example: Calling the argument constructor...

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

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, Declaration:

```
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
```

new

If a class definition includes a call to new, you *must* include a copy constructor.

- Or else the pointers will not behave correctly.
- Remember: the new operator returns a pointer.

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 members)

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**" or "**protected**" declaration allows this.

Compiling & Templates

Template code cannot be built into a .o object file by itself.

- Do NOT #include the .h header file at the top of the .cpp implementation file.
- Rather: #include the .cpp implementation file at the bottom of the .h header file.
- In the .h header file, just above the #endif, #include the .cpp implementation file (effectively making the template code one file to be #included at the top of the main/driver.cpp program.

Problems:

1. This is poor coding practice (not supposed to #include .cpp files in header files).
2. If you are using a makefile: Any file that requires a template class file must include, in its dependencies, both the header AND the implementation file.

Function Templates

1. No declaration:
Many compilers do not support template function declarations or separate compilation. Therefore, do NOT use template function declarations.

2. In same file:
Place the template function definition in the same file in which it is used.

3. Definition before Call:
Place the template function definition before it is called (i.e., before main()).

Writing Function Templates

1. Write a version of the function that is NOT a template.
2. Completely test and debug it.
3. Convert that ordinary function to a template function by adding the template prefix and replacing the type names with the type parameter.

Template Prefix: (must prefix the function definition)

```
template<class TypeParameter>
  or
template<typename TypeParameter>
```

Note: *no semicolon following the template prefix!*

Definition, example: (swap two values)

```
template<class ItemType>
void swap<ItemType &var1, ItemType &var2)
{
    ItemType tmp;

    tmp = var1;
    var1 = var2;
    var2 = tmp;
}
```

Call, example: (swap two values)

```
int first(1), second(2);
swap(first, second);
  or
char first('a'), second('b');
swap(first, second);
```

Class Templates

Template Prefix: (must prefix the class definition)

```
template<class TypeParameter>
  or
template<typename TypeParameter>
```

Note: *no semicolon following the template prefix!*

Definition, example:

```
template<class ItemType>
class Pair
{
  public:
    Pair(ItemType val_1, ItemType val_2);
    void set_item(int position, ItemType val);
    void get_item(int position) const;
  private:
    ItemType first;
    ItemType second;
};
```

Class Template Constructor:

```
template<class ItemType>
Pair<ItemType>::Pair(ItemType val_1, ItemType val_2)
{
   first = val_1;
   second = val_2;
}
```

Class Template Member Functions:

```
template<class ItemType>
void Pair<ItemType>::set_item(int position,
                              ItemType val)
{
   if (position == 1)
       first = value;
   else if (position == 2)
       second = value;
   else
       cout << "Error";
}
```

Class Object Declaration:

1. Append a <*type argument*> to the class name.
2. Follow that with the variable name and initial values.

Examples:

```
Pair<int> numbers(1, 2);
Pair<char> letters('a', 'b');
Pair<std::string> words("hi", "there");
```

See table in left column with notes on compiling issues when using template classes.

Arrow Operator ->

Specifies a member of a struct or 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;       // ptr var of type Record
p = new Record;  // ptr assigned to var p
p->number = 20;  // assign value to prop
p->grade = 'A';  // assign value to prop
```

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.

Operators: (*this) and this->

```
Point Point::move(double xC, double yC) const
{
  Point answer;

  answer.setX( getX() + xC );
      //  getX will be called by the same invoking
      //  instance that called the move function
      // *** OR ***
  answer.setX( (*this).getX() + xC);
  // "this" holds the address of the invoking instance

  answer.setY( getY() + yC );
      // *** OR ***
  answer.setY( this->getY() + yC);
  // this->  is the same as (*this).

  return answer;
}
```

Virtual Methods

Usage:
Used when a function will have a different definition in a derived class than in the base class.

In the base class, prefix the method declaration with the **keyword virtual**.

- The keyword is added to the declaration, NOT the definition.
- You only need to add virtual to the base class declaration (not to subsequent derived class declarations).
- However, for readability, it is good practice to add the keyword to derived class declarations.

The code virtual methods execute is determined at runtime (not when compiled) based on the calling object (which class it is from).

Constructors ***cannot*** be virtual.
Destructors ***can*** and ***should*** be virtual.

- This ensures future descendants of the object can deallocate themselves correctly

Example Syntax:

```
virtual void setItem(const ItemType& theItem);
```

Pure Virtual Methods: =0

The **=0** is a type of **keyword** syntax to denote a **pure virtual method**.

A pure virtual method (a virtual method that has =0 on the end) cannot be implemented by the base class.

- It must be implemented by a derived class.

Example Syntax:

```
virtual void setItem(const ItemType& theItem) =0;
```

Polymorphism

This delayed decision about which code to execute is an example of polymorphism

Polymorphism is the ability to associate multiple meanings to one function name by means of late binding (i.e., binding the meaning to the function name "later," at runtime instead, of when compiled).

Invoking Operators

operator notation	function notation
a + b	a.operator+(b)
a = b	a.operator=(b)

Note the use of the keyword **operator**.

Overloading Operators

Concept: An operator (like + or - or *) is just a function with a different syntax.

(a + b) is the same as **a.operator+(b)**

Syntax: Use the name of the operator (+) instead of a function name (e.g., add) and precede the operator with the **keyword "operator"**.

Binary: The parameters of the function are the left and right operands (a + b).

Unary: One parameter/argument is given: it is the right (and only) operand.

Example, Declaration (similar syntax for binary):

```
class Money
  {
    friend Money operator-(const Money &amt);
    // overload negative sign
    ...
  }
```

Example, Definition:

```
Money operator-(const Money &amt)
{ ...routine to return Money object as negative... }
```

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 (i.e., *copy right side to left side*):
- 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

a = b; same as **a.operator=(b);**

Declaration (in class definition):

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

(*can return void or a reference to the calling object*)

Definition (in implementation file):

```
Class_Name& Class_Name::operator=(const Class_Name
                                  &right_side)
{
   ... copy right_side.variables to calling obj...
   return *this // for function chaining
}
```

Overloading: Binary Examples (+)

```
Number a(345), b(12678); // Assume Number is a class
```

Example: object addition (a + b)

Declaration:

```
Number operator+(const Number &r) const;
```

Defintion:

```
Number Number::operator+(const Number &r) const
{
   return Number( getInt() + r.getInt() );
   // OR:
   return Number( (*this).getInt() + r.getInt() );
   // OR:
   return Number( this->getInt() + r.getInt() );
}
```

Call/Usage:

```
operator notation: a + b
function notation: a.operator+(b)
```

Example: mixed mode, int on right (a + 7439)

Declaration:

```
Number operator+(int n) const;
```

Defintion:

```
Number Number::operator+(int n) const
{
   return Number( getInt() + n );
   // OR:
   return Number( (*this).getInt() + n );
   // OR:
   return Number( this->getInt() + n );
}
```

Call/Usage:

```
operator notation: a + 7439
function notation: a.operator+(7439)
```

Example: mixed mode, int on left (567 + a)

Declaration:

```
friend Number operator+(int n, const Number &r );
```

Definition:

```
Number operator+(int n, const Number &r )
{
    return r + n;
    // The order is specific: object first. This
    // calls overloaded operator: r.operator+(n),
    // mixed mode, int on right. Then it returns
    // the Number object returned by the
    // overloaded operator.
}                          [NOTE: friend function]
```

Call/Usage:

```
operator notation: 567 + a
function notation: operator+(567,a)
```

Overload Insertion (Output) Operator: <<

Binary operator that returns the indicated output stream.

- When you return a stream, you **MUST** return a **REFERENCE** to that stream: & (which means you are returning the object itself, not the value of the object).
- The parameters of the function are the left and right operands.

Example, Declaration:

```
class Money
{
  public:
  friend ostream& operator<<(ostream& outs,
                             const Money &amt); ...
}
```

Example, Definition:

```
ostream& operator<<(ostream& outs, const Money &amt)
{
   ...output routine...;
   return outs; // returns a reference
}
```

Overload Extraction (Input) Operator: >>

Binary operator that returns a reference to the indicated input stream.

- The second parameter/argument receives the value piped in from the first parameter/argument.

Example, Declaration:

```
class Money
{
  public:
  friend istream& operator>>(istream &ins,
                             Money &amt);  ...
}
```

Example, Definition:

```
istream& operator>>(istream &ins, Money &amt)
{
   ...input routine...;
   return ins; // returns a reference
}
```

Assertions

```
#include <cassert> // include direction for assert()
```

assert() syntax

```
assert( someBooleanCondition );
```

if true: program continues normally
if false: program halts, error displayed

The assert() function is a **debugging tool**, not a substitute for exceptions.

Exceptions

```
#include <stdexcept> // include directive, exceptions
```

Two Parts of Working with Exceptions:

1. **Throw (inside)**
   - Inside a function that may have an error
   - Inside the function, throw if error
   - Inside an if-statement checking for an error
2. **Try/Catch (outside)**
   - Outside the function
   - "Try" to call the function
   - "Catch" any exceptions thrown by the function

1. **Inside function:** throw exceptions
2. **Outside function:** handle exceptions
   a. Try function
   b. Catch exceptions

noexcept

Mark a function "noexcept" if you are 100% sure it will not throw and exception.

Syntax: Append the keyword noexcept to end of function header.

```
void SafeFunction( int var ) noexcept;
```

Without the keyword, the function MAY throw an exception

NOTE: The use of "specifiers" (a throw clause in a function header) is **deprecated** as of C++11.
- If a function will NOT throw an exceptions, mark it with the noexcept keyword.
- Otherwise, if it MIGHT throw an exception, leave it

Exception Class Hierarchy

```
exception
 |
 |-> logic_error: internal logic error
 |    |
 |    |-> invalid_argument: invalid function argument
 |    |
 |    |-> length_error: object too large
 |    |
 |    |-> out_of_range: index out of range
 |-> runtime_error: error at runtime
      |
      |-> overflow_error: math, number too large
      |
      |-> range_error: result cannot be represented
      |
      |-> underflow_error: math, number too small
```

Custom Exception Class

Inherit from exception or any of its children

```
class SpecExcept : public logic_error
{
    public:
    SpecExcept(const string& what_arg)
               : logic_error (what_arg);
  {
    // do special things here
  }
};
```

Throw it from a function

```
float RiskyFunction( int num, int denom )
{
   if ( denom == 0 )
     throw SpecExcept( "0 denominator" );
   return float(num) / float(denom);
}
```

Try and Catch

```
try
{
  float result = RiskyFunction (num, denom);
  cout << "Result: " << result << endl;
}
catch ( SpecExcept e )
{
  cout << "ERROR: " << e.what() << endl;
}
```

Output with error:

```
"ERROR: 0 denominator"
```

throw

throw syntax

```
throw ExceptionClass( "string argument" );
```

throw example

```
// function that might receive an invalid argument
void ClassName::SetFunction( int num, string name )
{
  if ( num < 0 || num > 100 ) // we want 1-100 only
      throw out_of_range( "Invalid input" );
  anArray[num] = name;
}
```

throw multiple

```
// function that might receive different types of
//  invalid arguments
void SomeFunction( int error )
{
  if ( error == 1 )
    throw logic_error( "Logic Error" );
  else if ( error == 2 )
    throw runtime_error( "Runtime Error" );
  else if ( error == 3 )
    throw out_of_range( "Out of Range Error" );
}
```

try/catch

try/catch explanation

1. If a called function throws an exception, the caller needs to catch it.
2. Exceptions are handled in a try/catch block.
   a. If any exceptions are thrown in the try block...
   b. The code in the catch block is executed.
3. The what() method returns a c-string of the error message.
4. If no method or function handles the exception thrown, the program terminates
5. Arrange multiple catch blocks in order of specificity (specific to general)

try/catch sytax

```
try
{
  // statements that might throw an exception
}
catch (ExceptionClass identifier)
{
  // statements that react to an exception of
  // type ExceptionClass
  // identifier = catch block parameter
  // this NOT a function; it resembles a function
}
```

try/catch example

```
try
{
  SomeFunction( num );      // may throw exception
}
catch ( invalid_argument& e ) // catch specfic first
{
  cout << e.what() << endl;
}
catch ( logic_error& e )    // catch more general
{
  cout << e.what() << endl;
}
catch( exception& e )       // catches any exception
{
  cout << e.what() << endl;
}
```

blank

Whatever