SPARKCHARTS™

C++

COMMENTS

In C++, comments are either single lines or blocks. Denote single line comments with the `//` operator. Denote a comment block by enclosing the block with `/*` and `*/`.

Single line comment example:

```
if ( x == 0) { // Single Line Comment
}
```

Comment block example:

```
if ( x == 0) {
/* This is a multiline
   comment using the comment block style.
   */
}
```

Either style of comment can start on a line containing code as long as the comment is after the code.

PRIMITIVE DATA TYPES

Primitive data types are data structures built into the C++ language. They include basic numeric data, such as integers and floating point numbers, as well as character and Boolean variables.

A. Primitive data types:
- `void`: generic identifier that does not imply type
- `int`: basic integer variable; 2 or 4 bytes
- `float`: basic floating point variable; 4 bytes
- `double`: double precision floating point variable; 8 bytes
- `char`: basic ASCII character variable; 1 byte
- `bool`: basic Boolean variable; 1 byte

B. Modifiers: All modifiers default to `int` if a base type is not specified.
- `unsigned`: does not use a sign bit
- `long`: may have double number of bytes as base type; implementation dependent
- `short`: may have half the number of bytes as base type; implementation dependent

C. Arrays: Arrays are blocks of consecutive data structures. Arrays are declared with a specific size in the following way:
`int a[size];`
A specific part of an array is referenced using an **index**, which is an integer value from 0 to `size - 1`.

D. Enumeration: A data type that holds a set of values defined by the user. An enumeration can be named, and if so, the name is also the type. Members of an enumeration are integral types that have values associated with them. The user declares the values in a list enclosed by brackets:
`enum cards {CLUBS, DIAMONDS, HEARTS, SPADES}`
The enumeration type is `cards`, and the integral types are assigned to the enumerator values as follows: `0==CLUBS, 1==DIAMONDS, 2==HEARTS, 3==SPADES`

BASIC CONTROL STRUCTURES

The basic logic of a program is captured within its control structures. Control structures provide the sequential framework of a program. All conditional tests are Boolean tests for `true` or `false`. It is possible to cast non-Boolean values, either implicitly or explicitly, to fulfill the requirements of a conditional test.

`if/else` **statement:** The most basic conditional test is an if conditional. If the `conditional test` of the `if` statement returns true, then `code` is executed. Add additional conditional statements either with `else` (a default case executed if no other `if` or `else if` tests are met) and with `else if` statements. During run-time, only the first `true` conditional in any `if/else` statement will have the associated code executed. Any number (including none) of `if/else` statements are allowed. If the optional code is a single statement, then opening/closing code braces are also optional.

```
if (conditional test) {
code }
else if (conditional test)
{ code}
...
else {code}
```

`switch` **statement:** Each set of cases is broken by a `break;` statement. Multiple cases can lead into the same set of code. Until a `break` is met the code continues to run, so a `break` statement should separate each set of distinct code segments. A `break` statement is optional after the default case of the `switch` statement. The `switch` variable is tested for equality with the test value.

```
switch (switch variable) {
case test value:
additional cases
code
break;
...
default:
default code
}
```

`while` **and** `do/while` **statements:** `while` statements either test then start execution or execute a single iteration of the code block before checking the loop test. If the code is run before testing, then it is a `do/while` statement. If the `while` loop is entirely contained in the loop test, opening/closing code braces are optional.

```
while (conditional test);
```

or

```
while (conditional test) {
      code
}
```

or

```
do {
      code
} while (conditional test);
```

`for` **loop:** `for` loops function like `while` and `do/while` loops, except that the initialization and increment statements are part of the `for` structure. Braces for self-contained loops or single-line loops are optional.

```
for (initialization
statement; conditional test;
increment statement);
```

or

```
for (initialization
statement; conditional test;
increment statement) {
code
}
```

`break` **and** `continue` **commands:** The commands `break;` and `continue;` are useful for altering the flow of control. If a `break` command is executed in a `while`, `do/while`, `for`, or `switch` statement, then the execution immediately leaves that control structure. If a `continue` statement is executed in a `while`, `do/while`, or `for` structure, the flow of control immediately skips the rest of the code in the loop. In the case of a `while` or `do/while` statement, the `continue` statement means that the program will execute the conditional test. In the case of a `for` loop, the increment statement is executed, after which the conditional test is checked.

```
while (conditional test)
{
    code
    continue; /* Will skip
    to conditional test */
    code
    break; /* Will
    immediately end loop
    execution completely */
    code
}
```

OPERATORS

Assignment operator

`a = b;`	Assigns the value of b to a. Assignments are done from right to left, which means that `a = b = c;` will give you a case where a and b both have the same value as c.

Comparison operators

`a == b`	Evaluates to `true` if a equals b.
`a < b`	Evaluates to `true` if a is less than b.
`a > b`	Evaluates to `true` if a is greater than b.
`a != b` or `a not_eq b`	Evaluates to `true` if a is not equal to b.
`a <= b`	Evaluates to `true` if a is less than or equal to b.
`a >= b`	Evaluates to `true` if a is greater than or equal to b.
`!a` or `not a`	`!` is the not operator. It changes 0 to 1 and all other values to 0, which equates to `true` becoming `false` and `false` becoming `true`.
`a && b` or `a and b`	And operator. If both a and b are `true` then the expression evaluates to `true`. Otherwise it evaluates to `false`.
`a || b` or `a or b`	Or operator. Evaluates to true if either a or b is `true`. If both are `false` it evaluates to `false`.

Arithmetic operators

`a + b`	Addition
`a - b`	Subtraction
`a * b`	Multiplication
`a / b`	Division
`a % b`	Modulus (integer remainder)
`a += b`	Addition and assignment (`a = a + b`)
`a -= b`	Subtraction and assignment (`a = a - b`)
`a *= b`	Multiplication and assignment (`a = a * b`)
`a /= b`	Division and assignment (`a = a / b`)

Increment and decrement operators

`a++`	Returns value of a, then increments a; also known as pre-fix increment.
`++a`	Increments a, and then returns the incremented value of a; also known as post-fix increment.
`a--`	Returns value of a, then decrements a; also known as pre-fix decrement.
`--a`	Decrements a, and then returns the incremented value of a; also known as post-fix decrement.

Bitwise operators

`a & b` or `a bitand b`	Bitwise and
`a | b` or `a bitor b`	Bitwise or
`a ^ b` or `a xor b`	Bitwise xor
`a &= b` or `a and_eq b`	Bitwise and combined with assignment (`a = a & b`)
`a |= b` or `a or_eq b`	Bitwise or combined with assignment (`a = a | b`)
`a ^= b` or `a xor_eq b`	Bitwise xor combined with assignment (`a = a ^ b`)
`~a` or `compl a`	Bitwise not (also known as complement)
`a << b`	Bitwise left shift. Bits shifted off the left end are lost and 0's are inserted into the vacated bits on the right.
`a >> b`	Bitwise right shift. Bits shifted off the right end are lost and 0's are inserted into the vacated bits on the left.

“TO ERR IS HUMAN, BUT TO REALLY FOUL THINGS UP REQUIRES A COMPUTER.”

ANONYMOUS

BASIC STREAM OBJECTS

The standard stream objects are all opened automatically and provide basic input and output features for C++ programs. They are manipulated using the stream insertion (<<) and extraction (>>) operators, depending on whether it is an input stream or an output stream.

`cin >> a;`	**cin – Console Input Stream**
`cout << “Here”;`	**cout – Console Output Stream**
`cerr << “Error Found”;`	**cerr – Error Output Stream**
`clog << “Error Found”;`	**clog – Standard Log Stream**

POINTERS

A pointer is a variable that contains the memory address of a data structure. In other words, `int* a` does not contain an integer value. Instead, `a` would contain a reference to a memory location that is an integer value. Pointer variables can also be used to reference arrays of variables and can be assigned and reassigned during run-time. Pointers can also be used to hold pointers to dynamically assigned memory. Pointers can also be assigned the value of `NULL`. `NULL` is a C++ keyword that represents an inaccessible memory location and is commonly used by programmers to represent an unassigned pointer.

Pointer declaration example:

```
int* a = NULL;
```

Pointers must be dereferenced before use. This dereferencing is accomplished with the `*` operator. The `&` operator can also be used on a nonpointer object to get a reference to that object, as in the example below.

Pointer assignment and dereferencing example:

```
char b = 'a';
char *a = &b;
cout << *a << endl;
```

DYNAMIC MEMORY ALLOCATION

During program execution, sometimes it is necessary for a program to allocate additional memory or to dynamically create or destroy objects. Use the `new` and `delete` keywords to dynamically allocate memory.

New and delete examples:

```
int* a = new int; // creates a pointer to a new integer
delete a; // delete memory at a
a = new int[15]; // create a new integer array of size 15
delete[] a; // Delete an array at memory location a
```

FUNCTIONS

Functions return at most a single value. Functions may also have a `void` return type, which means the function has no value returned. The values needed by the function are called **parameters**, and some or all of these parameters may be assigned default values. Functions may also have no parameter list.

Function format:

```
return_type function(type parameter=default value, type
parameter2=default value...)
{
      code
}
```

Functions that have not yet been declared will not be recognizable to compilers. As such, it is necessary to use a **function prototype** earlier in the code to describe the requirements of a function that will appear later. Function prototypes have the return type, function name, and parameter types.

Function prototype:

```
return_type function(type1, type2...);
```

To return a value, the `return` command is used. A function with a declared return type must return a value of the appropriate type. A function with a `void` return type may never return a value. The example below returns `x`, assuming that `x` is of the appropriate type.

Return statement:

```
return x;
```

Parameters are passed by either value, reference, or pointer. Parameters passed by value may be changed without affecting the original values of the passed variable. Variables passed by reference or pointer will retain any changes made to the original value when the function returns. A parameter passed by value uses the standard type identifier and is treated exactly like the base type. Parameters passed by reference have the `&` symbol used after the type name and are treated as the base type. Pass by pointer `parameters` use the `*` symbol after the type name and are treated as pointers of the base type.

Parameter passing example:

```
return_type function( type PassedByValue, type &PassedByReference,
type &PassedByPointer)
```

It is also possible for multiple functions to have the same function name. To resolve the possible confusion, the compiler will look at the call parameters of the functions available and choose the function with the closest match, if one is available. The method of using multiple functions of the same name is called function overloading.

Function overloading example:

```
void function(int a) { }
void function(bool b) { }
```

INCLUDE FILES

A programmer can include multiple additional files in a C++ program. These files, called include files, are used to access libraries or to break a program into multiple parts. For defined files of the C++ standard library, such files are included using the `#include` statement and the name of the file excluding the `.h` part of the file name.

iostream include statement example:

```
#include <iostream>
```

It is also possible to include other files that are not part of the standard libraries. These are included using the following style of statement:

Non-standard library include statement:

```
#include “includefile.type”
```

Note that recursive inclusions will cause problems and should be avoided.

NAMESPACES

Part of the C++ standard is the inclusion of namespaces. A namespace is used to distinguish libraries and sections of code from one another. Namespace usage helps prevent function name confusion if more than one function has the same name and parameter list. The namespace defines a scope where global identifiers and variables are placed. To define a namespace the following structure is used:

Namespace format:

```
namespace Name {
      code
}
```

Making use of code defined in a namespace requires either an explicit call to the namespace, or declaration before the name is used.

Namespace usage examples:

```
namespace Name {
      code
}
Name::member;
using namespace Name;
member;
```

The most common namespace is the C++ standard library namespace `std`.

PREPROCESSOR DIRECTIVES

Preprocessor directives can be used to define values, include or exclude portions of code, and perform other simple processing at compiler time. Examples of use for preprocessor directives include debugging code and preventing file recursion. They are also used to create macros or to provide various implementation-dependent messages.

The `#define` statement creates a symbolic name, which is then replaced within the code with a given set of replacement text. `#define` can be used to define constants, though the preferred method in C++ is to use a `const` object of the appropriate type. It can also be used to define macros.

#define format:

```
#define SymbolicIdentifier ReplacementString
```

Wherever the preprocessor finds the `SymbolicIdentifier`, it replaces it, before compilation, with the `ReplacementString`. By using a function style as the `SymbolicIdentifier` it is also possible to create macros, where a value can be used and replaced into the macro.

Macro example:

```
#define Macro( x ) (5 * x)
```

In the example, `Macro( x )` will be replaced in the program with `(5 * value_used_for_x)`. For example, if the code used `Macro( a )` then the preprocessor would replace it with `(5 * a)`. `Macro(7)` would be replaced with `(5 * 7)`. It is possible to use any number of arguments as part of the macro definition.

It is also possible to undefine a directive. Using `#undef` will remove the directive from the scope of the preprocessor. A directive also leaves scope at the end of the file.

#undef example:

```
#undef Macro( x )
```

It is also possible to use `#if`, `#ifdef` and `#ifndef` to manage conditional compilation of code. A block of code contained within a `#if`/`#ifdef`/`#ifndef` and `#endif` precompiler directive will be compiled only if the respective expression is true. `#ifdef` name and `#ifndef` name are used to represent `#if defined(name)` and `#if !defined(name)`, respectively.

#if constructions:

```
• #if expression
        code
  #endif
• #ifdef name
  code
  #endif
• #ifndef name
  code
  #endif
```

It is also possible to use `#` and `##` in macros. The `#` character will replace the argument with the string used for the macro argument surrounded by quotes.

# example:

```
#define Macro(x) cout << #x
Macro(Example); // Expands to cout << “Example”;
```

The `##` operator concatenates two tokens and must have two operands.

## example:

```
#define Macro(x,y) cout << “x##y”
Macro(Concat,Example); // Expands to cout << “ConcatExample”;
```

There are also 5 predefined symbolic constants; they cannot be redefined using `#define` or undefined using `#undef`

Predefined symbolic constants:

• `__LINE__`	The line number of the current line of code
• `__FILE__`	The name of the source file
• `__DATE__`	The date of compilation
• `__TIME__`	The time of compilation
• `__STDC__`	The integer 1. It is used to signify that the implementation is ANSI compliant.

CONTINUED ON OTHER SIDE

STRUCTS

`structs` are simple aggregate data structures that bind multiple types into a single structure that is recognized by the compiler.

Struct declaration:

```
struct StructureName {
type Name;
...
};
```

The members of a struct are referenced using the dot operator (`.`) if it is an instantiation of the struct. If it is a pointer to the struct, then use the arrow operator (`->`) instead. Quick tip: `x->y` is equivalent to `(*x).y`

Struct access examples:

```
StructureName.member;
StructurePointer->member;
```

CLASSES

Classes are advanced data structures. Unlike structs, classes contain functions associated with the data type. They also include specific functions for the creation and destruction of a class and a way to specify member access rights. A member function defined in a class definition is automatically inlined. To set a member function defined outside of a class definition as inline, use the keyword `inline`.

CONSTRUCTORS AND DESTRUCTORS

The creation and destruction methods are called constructors and destructors. They differ from normal functions in restrictions on return types, parameters, and naming conventions.

Constructor: A constructor never has a return type, can never return a value, and will always have the same name as the class itself. A constructor with no arguments is called the **default constructor** and will be called any time an instance of the class is created. By the nature of classes, if no default constructor is specifically described by the coder, a default one, which allocates memory for the respective data members, is assumed by the compiler and will be placed in the public space of the class. Explicitly declare default constructors for any class that makes use of pointers to prevent memory problems. There are also other types of constructors such as a **copy constructor**, which is used to make a copy of a class. The `explicit` keyword can be used to prevent the compiler from using a constructor to cast another object to the object using the explicit constructor.

Destructor: The destructor method uses the class name, preceded by the ~ character. There is only one destructor method for any given class, and a destructor never takes any parameters. The destructor function has no return type. Destructors are used by the programmer to clean up after the class. Whenever an instance of the class is freed from memory, the destructor is invoked to execute any cleanup required by the programmer.

MEMBER ACCESS SPECIFIERS

Each of the three member access specifiers provides a different level of access control to the data members and member functions of a class. The **public specifier** allows access to any outside source. The **private specifier** allows access only to other member functions and friend classes or functions. The **protected specifier** limits access to derived classes and their friend classes and functions. The specifiers can be declared in any order and any number of times.

Class declaration:

```
class Classname {
public: // Public Members
     Classname() // Default Constructor
     {
               code
     }
     Classname(Classname &C) //Copy Constructor
{
code
}
      explicit Classname(OtherClass &C) //Explicit Constructor
{
code
}
     ~Classname() //Destructor
private: // Private Members
     code
protected: // Protected Members
     code
};
```

OPERATOR OVERLOADING

Use operator overloading to replace existing behaviors of a class or to specify a method for an object with no predefined method for a given operator. In order to overload the `()`, `[]`, and `->` operators, you must use a class member. You can overload other operators using non-member functions if the leftmost operand is of the same class type. If it is of a different type, then you must overload the operator as a non-member function. If that operator function needs to access private or protected members, then the operator should also be declared as a `friend` function.

Examples of member and non-member overloaded operators:

```
class Classname {
public:
     bool operator==(Classname &c) /*Function for a Member
Operator*/
     {
               code
     }
     friend istream &operator>>(istream&, Classname &); /*
Function Prototype for a Friend non-member function*/
};
istream &operator>>(istream &is, Classname &c) /*Function for
non-member friend overload*/
{
     code
}
bool operator==(Classname &c_one, Classname &c_two) /*Non-member
operator overload*/
{
     code
}
```

CONST OBJECTS AND METHODS

The `const` keyword is used to identify objects that should not have their values modified. It is also used to identify methods that do not modify the contents of parameters passed to them. After initialization, it is not possible to modify a `const` object. It is only possible to call `const` methods and members of a `const` object during run-time.

const object examples:

```
const int a;
const Classname object;
```

A function declared `const`, either as a member function of a class or separate from a class, cannot modify any data members. Also, such functions can only call `const` functions internally.

const function examples:

```
class Classname{
     int classConstFunc() const;
     code
};
int nonClassConstFunc() const {}
```

MUTABLE CLASS MEMBERS

`mutable` class members are the opposite of `const` class members. A class member declared `mutable` is always modifiable, even in a `const` member function or `const` object.

Mutable example:

```
class Classname{
     mutable int mutableInt;
     code
};
```

INHERITANCE

Classes can derive features from one or several base classes. This capability allows the construction of class libraries with similar features and the ability to use multiple classes using one set of functions.

ACCESS PROTECTION SPECIFIER

You must specify an **access protection specifier** for a base class. This specifier determines which portions of the base class are visible to the **derived class**, and the level of protection for those portions. While `public` and `protected` members of the base class may change their protection in a derived class, all `private` members are hidden in the derived class and can be accessed only by member and `friend` functions through the use of `public` or `protected` members of the base class.

INHERITANCE SPECIFIER

With **public inheritance,** `public` and `protected` members of the base class are `public` and `protected`, respectively, in the derived class. With **protected inheritance,** both `public` and `protected` members of the base class are `protected` in the derived class. With **private inheritance,** both `public` and `protected` members of the base class are `private` in the derived class. C++ also allows multiple inheritance, in which a class can inherit more than one direct base class.

Inheritance examples:

```
class Base{
};
class SingleBase : public Base {
};
class BaseTwo {
};
class MultipleBase : public Base, public BaseTwo {
};
```

TEMPLATES

Templates are a way to provide a single method or set of methods that will be used by multiple types. Templates provide a powerful method for building and designing generic and reusable functions and classes. Templates also can be used to build generic class hierachies that allow for more degree flexibility. Templates functions and classes leave one or more of the contained types unspecified. The parameters, called **template parameters,** are then used throughout the function to represent the necessary types. The template parameters are treated exactly like a type.

Template function example:

```
template <class T>
T function( T variable)
{
     code
}
```

Multiple templates class example:

```
template <class T, class S>
class Classname {
public:
     T variableOne;
     S variableTwo;
code
};
```

Template parameters for classes are mostly used to declare objects that can hold multiple types. An example is the vector class from the C++ Standard Template Library (STL). The `vector` class implements a standard variable length array type that contain any basic or user defined data type.

```
int main() {
vector<int> intVector;
vector<char> charVector;
intVector.push_back(9);
intVector.push_back(1);
charVector.push_back('A');
return 1;
}
```

As can be seen from the example, a `vector` is given a specific type and then operates on that type. This is extremely useful for any storage data structure. Some storage classes may require additional abilities, such as a comparison operator, if they perform some action on the stored type. One such class is the `set` class, which is also part of the STL library. This class keeps objects in the same order and as such requires that a comparison operator either be defined for the class or given to the `set` class as part of the template arguments.

TEMPLATE SPECIALIZATION

Another use for templates is the implementation of generic functions or algorithms. Any generic algorithm can make use of this feature and still define specific implementations that optimize the algorithm for a given set of classes. A programmer can specify a generic implementation of a function for all classes, and then define specific implementations of the same function for a set of classes.

```
    template <class T>
    void Algorithm(vector<T> &vec_one) {
          //Some algorithm over the vector of objects
          code
    }
    template<>
    void Algorithm(vector<int> &vec_one) {
          // Same algorithm, but specific to integers
          code
    }
```

When a specific templated function or class is written and overloaded with specific arguments it is called **specialization.** Specialization can be either full or partial, depending on whether any of the original template parameters are left unbound. A good example of this is a hashtable class. A hashtable has both a stored data structure and a key. The types of the stored structure and the key can be completely different, so a generic implementation of a hashtable might look something like this:

```
    template <class Data, class Key>
    class hashtable{
          code
    };
```

Full specialization

Someone implementing or designing the hashtable class may wish to specify implementations of the class which have a given set of arguments, such as an implementation where the key is a `char` type and the data is a `String`. This specialization would look something like this:

```
    template<>
    class hashtable<String,char> {
          code
    };
```

Partial specialization

It is also possible to only implement some of the template parameters. This is called partial specialization and allows for the ability to optimize a class over one of its parameters. This example partially specializes `hashtable` with a float.

```
    template<class Key>
    class hastable<float, Key>{
          code
    };
```

NON-TYPE TEMPLATE PARAMETERS

Fully specialized templates are matched before partially specialized templates. Additionally, a partially specialized template will be matched before the unspecialized template is used.

Template parameters are not required to have types. A programmer can use non-type parameters as a method of setting up values for a class. One primary use of this style of template parameter is to define the sizes of a variable sized data structure, such as a list, matrix or table. In classes, which use a non-type parameter, a primary parameter would be the stored class type, while the second parameter might be the default size or actual size of the data structure. So if we wanted a matrix that was 50 by 100 and used integer values we might create a matrix class with the following statement:

```
    matrix<int,50,100> matrix_one;
```

And the matrix class would look something like this:

```
    template <class T, int rows, int columns>
    class matrix{
          /* Internal representation of a 2 dimensional static matrix
    with the given size.*/
          code
    };
```

TEMPLATE PARAMETER DEFAULTS

This additional information is then useable in any manner within the class as though it were a static type, even though the actual value may be set only at run-time.

A default can be specified for a template parameter, much like those specified for function arguments. This method of using template parameters becomes useful for defining a single implementation of a function or class, which may allow for a variable number of arguments, where some of the arguments may have default values associated with them. If, for example, the matrix class described above wanted to have a default row and column size of 10, we would see the class code looking something like this:

```
    template <class T, int rows=10, int columns=10>
    class matrix{
          /* Internal representation of a 2 dimensional static matrix
    with the given size.*/
          code
    };
```

Early declarations of a template parameter can be used to nest descriptions. This means that a programmer can write

```
    template <class T, class Comp = less<T> >
```

as part of the template header for a comparison algorithm. This helps simplify the task of the programmer by providing a way to build a template list based on earlier parameters.

NESTED TEMPLATES AND TEMPLATE PARAMETERS

You can also nest template declarations. A programmer can, for example, create a `vector` of `vectors` of `ints` or a hash table that uses a `vector` of `ints` as the key and a `set` of `Strings` as the stored values. Nesting templated classes is done simply by nesting the given classes within the correct template parameters, as in the examples below:

```
    vector< vector< int> > vec_of_vec_of_int;
    hashtable< vector<int>, set<string> > hash_table;
```

POLYMORPHISM, VIRTUAL FUNCTIONS

- The idea of **polymorphism** is that individual classes can and will respond differently to the same request. Polymorphism makes it possible to design a standard interface that all classes in the hierarchy implement differently, based on the needs of each class.
- One example of this is a **Shape library.** All classes in the library could inherit a `draw` function from an abstract class, called `Shape`. `Shape` would set up the parameters for the function, and an application might store an array of `Shapes`. This array would actually have multiple different classes which are all children of `Shape`, such as `Rectangle` and `Circle`. A call to each of the `draw` functions for each `Shape` object in the array could then be done without knowing specifically what kind of object had been used to implement the `Shape` base class. Instead, the class-specific `draw` function for each shape would be called and implemented according to the specific shape.

- In C++, **virtual functions** are used to implement polymorphism. Virtual functions are functions that are specified in the base class of an object hierarchy and then passed on to the derived classes. Any derived classes then implement a version of the virtual function specific to their needs. Object member functions are normally matched with the current cast of the object type during run-time; however, in the case of virtual functions, the cast type is ignored and the function originally instantiated with the object is called. It is therefore possible to use a single interface on multiple types of objects and have the correct functions executed on a given object.
- A virtual function that is not actually implemented in the base class is a **pure virtual function.** It is specified by setting the function prototype equal to 0 in the base class. This is used to set up interfaces that do not have any default implementation, and requires that any derived class define its own specific version of the virtual function. Using virtual functions is different from overriding a base class's method because the cast of the object does not affect which method is executed. If the function is merely overloaded in a derived class, it is possible to execute the base class method if the object is cast as an object of the base type.

Virtual function example:

```
    class Classname {
          virtual void pureVirtualFunc() const = 0; //Pure virtual
    Function
          virtual void virtualFunc()    // Virtual Function
    {
                code
    }
    };
```

An abstract class is a base class with at least one virtual function. Abstract classes are useful as interfaces and provide a framework for derived classes in a class hierarchy. The `Shape` class in the earlier example would be a good case for an abstract class. The `Shape` library described might look something like this:

```
    class Shape{
          public:
          Shape() {}
          virtual void draw() const = 0; //Pure Virtual Function
    };
    class Rectangle : public Shape {
          public:
          Rectangle() {}
          void draw() { /* Draws a rectangle */ }
    };
    class Circle : public Shape {
          public:
          Circle() {}
          void draw() { /* Draws a circle */ }
    };
```

Combining abstract classes with templated data structures and functions provides a powerful method for eliminating object-specific implementations outside of the objects themselves. Different implementations of abstract classes are completely transparent to the end user.

CAST OPERATORS

Cast operators are introduced into the ANSI C++ standard to replace old C style casting. All cast operators use the style `cast_type_<class>(object)` where `class` is the class to cast to, and where `object` is the object, pointer, or reference to be cast.

`static_cast`	Conversion between types that is checked at run-time. Standard conversions.
`const_cast`	Used to cast away `const` or `volatile` specifiers so that an object can be modified as normal.
`dynamic_cast`	Used to move through a class hierarchy. It can cast a pointer or reference to any of its base classes or any other derived class, or it can cast a base class as a derived class.
`reinterpret_cast`	This type of casting is used to do non-standard casting or standard casts.

KEYWORDS

`and`	`dynamic_cast`	`operator`	`try`
`and_eq`	`else`	`or`	`typedef`
`asm`	`enum`	`or_eq`	`typeid`
`auto`	`explicit`	`private`	`typename`
`bitand`	`export`	`protected`	`union`
`bitor`	`extern`	`public`	`unsigned`
`bool`	`false`	`register`	`using`
`break`	`float`	`reinterpret_cast`	`virtual`
`case`	`for`	`return`	`void`
`catch`	`friend`	`short`	`volatile`
`char`	`goto`	`signed`	`wchar_t`
`class`	`if`	`sizeof`	`while`
`compl`	`inline`	`static`	`xor`
`const`	`int`	`static_cast`	`xor_eq`
`const_cast`	`long`	`struct`	
`continue`	`mutable`	`switch`	
`default`	`namespace`	`template`	
`delete`	`new`	`this`	
`do`	`not`	`throw`	
`double`	`not_eq`	`true`	

SPARKCHARTS™

Writer: James Kilbride
Design: Dan O. Williams
Series Editor: Sarah Friedberg

$4.95 | $7.95 CAN