C++

Memory Management

The keyword `new` replaces `malloc` and `delete` replaces `free`

```
char* s = new char[size];   // dynamically allocate memory for an array
delete [] s;                // free the allocated memory
s = nullptr;                // good practice for preventing errors
```

Pointers

Constant values & constant pointers

```
// this function accepts a pointer to an array of constants
void displayPayRates(const double* rates, int size)
{
    for (int count = 0; count < size; count++)
    {
        cout << rates[count] << endl;
    }
}

// constant pointers can not point to something else
int value = 22;
int* const ptr = &value;

// this is a constant pointer to a constant
int number = 15;
const int* const ptr = &number
```

Pass by Reference

Swap in C

```
// i and j are pointers to ints
void swap(int* i, int *j)
{
    int temp = *i; // dereference i
    *i = *j;
    *j = temp;
}

// have to pass addresses of a and b
swap(&a, &b);
```

Swap in C++

```
// i and j are references to ints
inline void swap(int &i, int &j)
{
    int temp i; // no need to dereference
    i = j;
    j = temp;
}

// C++ allows function overloading unlike C
inline void swap(double &i, double &j)
{
    double temp i;
    i = j;
    j = temp;
}

// no need to pass addresses
swap(a, b);
```

Vectors

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

vector<int> numbers1;                    // an empty vector of ints
vector<int> numbers2(10);                // a vector of 10 ints
vector<int> numbers3(10, 2);             // a vector of 10 ints, each initialized to 2
vector<int> numbers4 {10, 20, 30, 40};   // a vector initialized with an initialization list
vector<int> myVec(numbers4);             // a vector initialized with the elements of numbers4

int val = myVec.at(index);  // return the value of the element located at index of myVec
int* arr = myVec.data();    // return the underlying int array of myVec
myVec.push_back(50);        // create a last element (if myVec is full) and stores 50 in it
myVec.pop_back();           // remove the last element from myVec
myVec.size();               // get the number of elements in myVec
myvec.capacity();           // get the capacity of myVec
myVec.clear();              // completely clear the contents of myVec
myVec.empty();              // return true if myVec is empty
myVec.reverse();            // reverse the order of elements in myVec
myVec.resize(size, val);    // resize myVec. the new elements are initialized with val
myVec.swap(someVec);        // swap the contents of myVec with the contents of anotherVec
```

Character Functions

```
#include <cctype>   // required for using the following functions
```

Character testing

Function	Returns true if the argument is a ...; returns 0 otherwise
`isalpha`	letter of the alphabet.
`isalnum`	letter of the alphabet or a digit.
`isdigit`	digit from 0 through 9.
`islower`	lowercase letter.
`isprint`	printable character (including a space).

Function	Returns true if the argument is a ...; returns 0 otherwise
`ispunct`	printable character other than a digit, letter, or space.
`isupper`	uppercase letter. Otherwise, it returns 0.
`isspace`	whitespace character. (`' '`, `' \n '`, `' \v '`, `' \t '`)

Character case conversion

Function	Description
`toupper`	Returns the uppercase equivalent of its argument.
`tolower`	Returns the lowercase equivalent of its argument.

C-Strings

Working with c-strings

```
#include <cstring>   // required for using the following functions
```

The `strlen` function

```
// don't confuse the length of a string with the size of the array holding it
char name[] = "Thomas Edison";
int length = strlen(name); // length is 13
```

The `strcat` function (see also: `strncat`)

If the array holding the first string isn't large enough to hold both strings, `strcat` will overflow the boundaries of the array.

```
char string1[100] = "Hello ";   // string1 has enough capacity for strcat
char string2[] = "World!";
strcat(string1, string2);
cout << string1 << endl;        // outputs "Hello World!"
```

The `strcpy` function (see also: `strncpy`)

`strcpy` performs no bounds checking. The array specified by the first argument will be overflowed if it isn't large enough to hold the string specified by the second argument.

```
char name[] = "Some other string";  // size of the holding array is sufficient
strcpy(name, "Albert Einstein");
```

The `strstr` function

```
char arr[] = "Four score and seven years ago";
char *ptr = strstr(arr, "seven");   // search for "seven" and return address
cout << ptr << endl;                // outputs "seven years ago"
```

The `strcmp` function

```
int strcmp(char *string1, char *string2); // function prototype
```

The result is

- **zero** if the two strings are **equal**.

- **negagive** if string1 comes **before** string2 in alphabetical order.
- **positive** if string1 comes **after** string2 in alphabetical order.

C-string/numeric conversion functions

```
#include <cstdlib>   // required for using the following functions

//Converts a c-string to an integer.
int intVal = atoi("1000");

//Converts a c-string to a long value.
long longVal = atol("1000000");

//Converts a c-string to a double/float value.
float floatVal = atof("12.67");
double doubleVal = atof("12.67");

// returns a numeric argument converted to a c-string object
to_string(int value);
to_string(long value);
to_string(long long value);
to_string(unsigned value);
to_string(unsigned long value);
to_string(unsigned long long value);
to_string(float value);
to_string(double value);
to_string(long double value);
```

Strings

Defining `string` objects

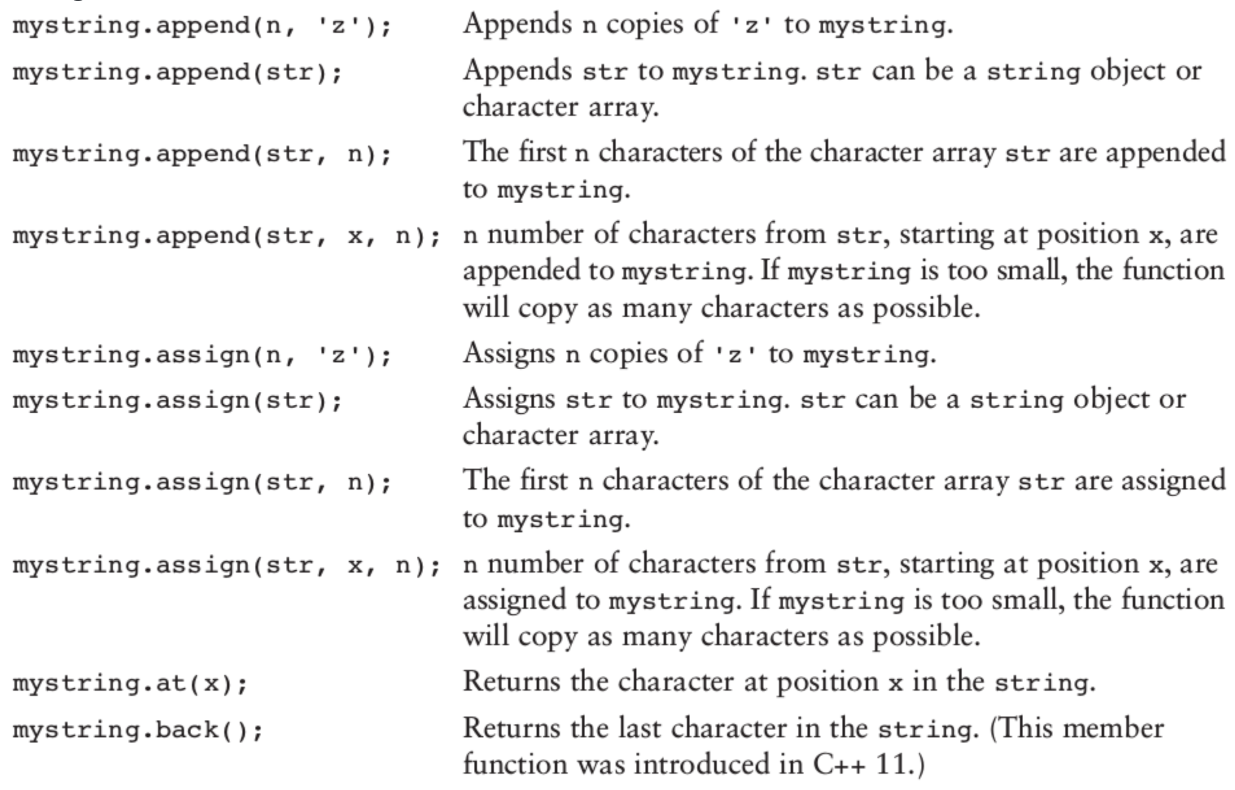
```
// required for using the string data type
#include <string>

// defines an empty string
string str0;

// defines a string initialized with "Hello"
string str1 = "Hello";

// defines a string initialized with "Greetings!"
string str2("Greetings!");

// defines a string which is a copy of str2. (str2 may be a string or a c-string)
string str3(str2);

// this has to be a c-string, not a string
char cStr[] = "abcdefgh";

// defines a string which is initialized to the first 5 characters in cStr
string str4(cStr, 5);

// defines a string initialized with 10 'x' chars
string str5('x', 10);

// defines a string which is initialized with a substring of str5.
string str6(str5, 2, 8);
```

`string` operators

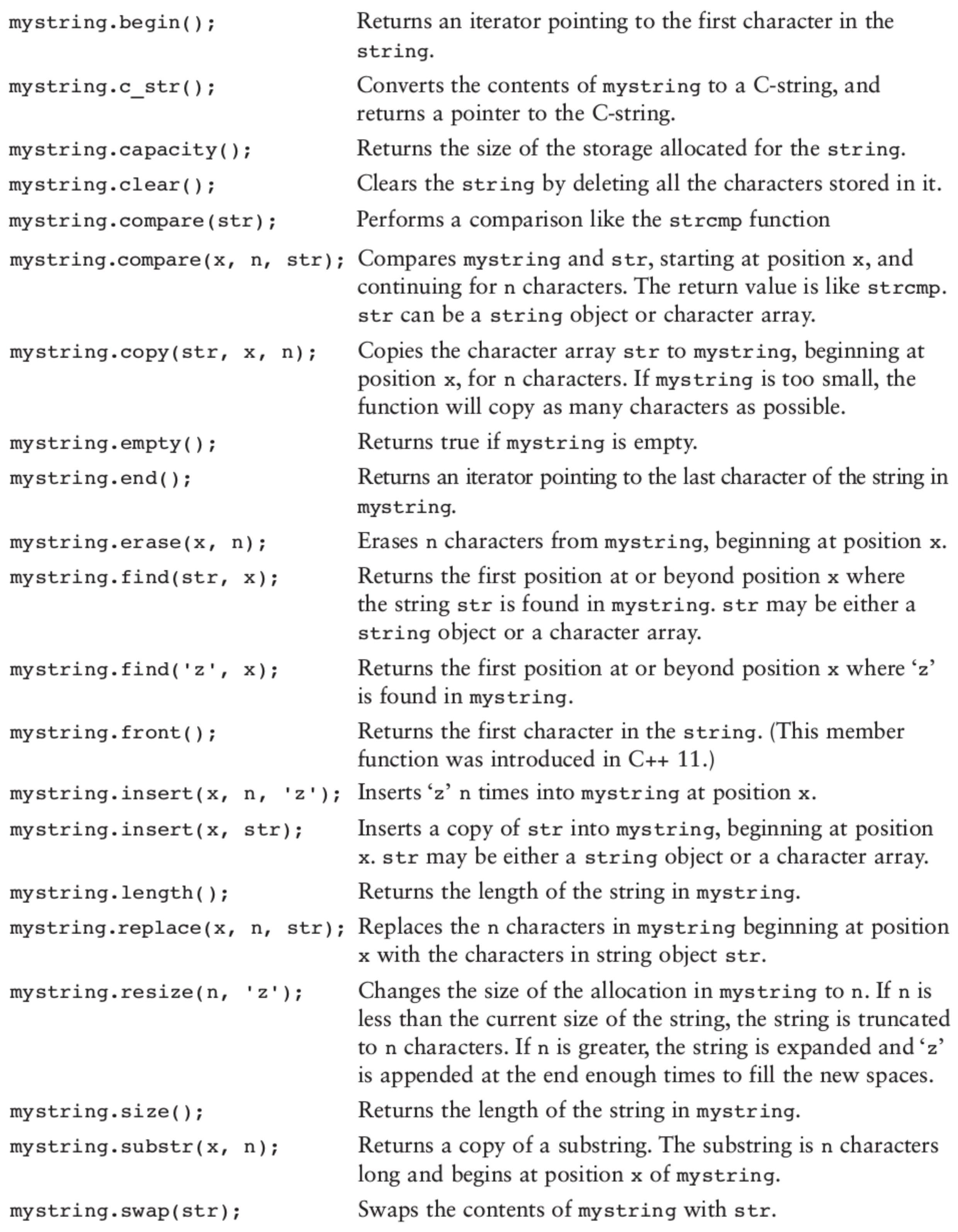
There is no need to use a function such as `strcmp` to compare string objects. You may use the `<` , `>` , `<=` , `>=` , `==` , and `!=` relational operators.

```
string s1 = "Hello ";
string s2 = "World!";
string mystring = s1 + s2;  // concatenates s1 and s2
char c = mystring[0];       // returns the char at position 0 in mystring
```

`string` member functions



`mystring.append(n, 'z');`	Appends `n` copies of `'z'` to `mystring`.
`mystring.append(str);`	Appends `str` to `mystring`. `str` can be a `string` object or character array.
`mystring.append(str, n);`	The first `n` characters of the character array `str` are appended to `mystring`.
`mystring.append(str, x, n);`	`n` number of characters from `str`, starting at position `x`, are appended to `mystring`. If `mystring` is too small, the function will copy as many characters as possible.
`mystring.assign(n, 'z');`	Assigns `n` copies of `'z'` to `mystring`.
`mystring.assign(str);`	Assigns `str` to `mystring`. `str` can be a `string` object or character array.
`mystring.assign(str, n);`	The first `n` characters of the character array `str` are assigned to `mystring`.
`mystring.assign(str, x, n);`	`n` number of characters from `str`, starting at position `x`, are assigned to `mystring`. If `mystring` is too small, the function will copy as many characters as possible.
`mystring.at(x);`	Returns the character at position `x` in the `string`.
`mystring.back();`	Returns the last character in the `string`. (This member function was introduced in C++ 11.)



`mystring.begin();`	Returns an iterator pointing to the first character in the `string`.
`mystring.c_str();`	Converts the contents of `mystring` to a C-string, and returns a pointer to the C-string.
`mystring.capacity();`	Returns the size of the storage allocated for the `string`.
`mystring.clear();`	Clears the `string` by deleting all the characters stored in it.
`mystring.compare(str);`	Performs a comparison like the `strcmp` function
`mystring.compare(x, n, str);`	Compares `mystring` and `str`, starting at position `x`, and continuing for `n` characters. The return value is like `strcmp`. `str` can be a `string` object or character array.
`mystring.copy(str, x, n);`	Copies the character array `str` to `mystring`, beginning at position `x`, for `n` characters. If `mystring` is too small, the function will copy as many characters as possible.
`mystring.empty();`	Returns true if `mystring` is empty.
`mystring.end();`	Returns an iterator pointing to the last character of the string in `mystring`.
`mystring.erase(x, n);`	Erases `n` characters from `mystring`, beginning at position `x`.
`mystring.find(str, x);`	Returns the first position at or beyond position `x` where the string `str` is found in `mystring`. `str` may be either a `string` object or a character array.
`mystring.find('z', x);`	Returns the first position at or beyond position `x` where 'z' is found in `mystring`.
`mystring.front();`	Returns the first character in the `string`. (This member function was introduced in C++ 11.)
`mystring.insert(x, n, 'z');`	Inserts 'z' `n` times into `mystring` at position `x`.
`mystring.insert(x, str);`	Inserts a copy of `str` into `mystring`, beginning at position `x`. `str` may be either a `string` object or a character array.
`mystring.length();`	Returns the length of the string in `mystring`.
`mystring.replace(x, n, str);`	Replaces the `n` characters in `mystring` beginning at position `x` with the characters in string object `str`.
`mystring.resize(n, 'z');`	Changes the size of the allocation in `mystring` to `n`. If `n` is less than the current size of the string, the string is truncated to `n` characters. If `n` is greater, the string is expanded and 'z' is appended at the end enough times to fill the new spaces.
`mystring.size();`	Returns the length of the string in `mystring`.
`mystring.substr(x, n);`	Returns a copy of a substring. The substring is `n` characters long and begins at position `x` of `mystring`.
`mystring.swap(str);`	Swaps the contents of `mystring` with `str`.

File Operations

Data Type	Description
`ifstream`	Input file stream. Can be used to read data from files.
`ofstream`	Output file stream. Can be used to create write data to files.

Data Type	Description
`fstream`	File stream. Can be used to read and write data to/from files.

`ifstream` and `ofstream`

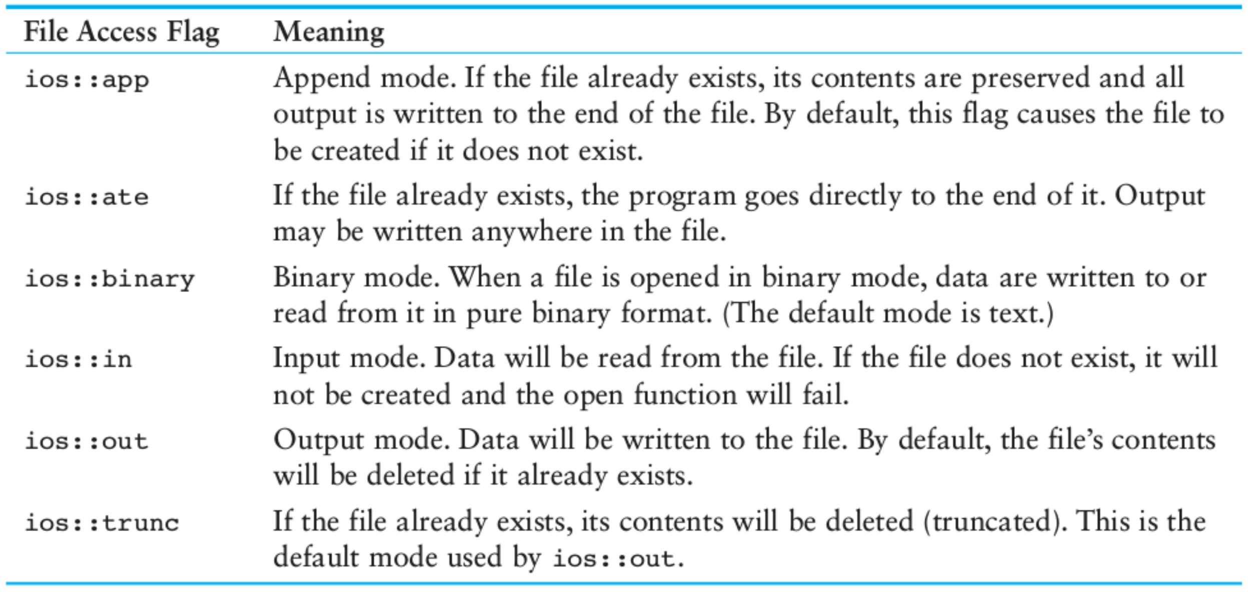
```cpp
#include <fstream>

// open an ifstream
ifstream inputFile;
inputFile.open("InputFile.txt");
// Alternatively:
// ifstream inputFile("InputFile.txt");

// open an ofstream
ofstream outputFile;
outputFile.open("OutputFile.txt");
// Alternatively:
// ofstream outputFile("OutputFile.txt");

// open ofstream in append mode
outputFile.open("OutputFile.txt", ios::out | ios::app);

inputFile >> value;
outputFile << "I love C++ programming" << endl;

inputFile.close();
outputFile.close();
```

`fstream`

```cpp
#include <fstream>

fstream file;

// open fstream in output mode
file.open("DataFile.txt", ios::out);

// open fstream in both input and output modes
file.open("DataFile.txt", ios::in | ios::out);

// read and write data in binary mode
char data[4] = {'A', 'B', 'C', 'D'};
fstream file("DataFile.dat", ios::binary | ios::out);
file.write(data, sizeof(data));
file.open("DataFile.dat", ios::binary | ios::in);
file.read(data, sizeof(data));

// read and write non-char data
int numbers[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
fstream file("numbers.dat", ios::out | ios::binary);
file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));
file.open("numbers.dat", ios::in | ios::binary);
file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));

// close the file
file.close();
```

File access flags

By using different combinations of access flags, you can open files in many possible



File Access Flag	Meaning
`ios::app`	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
`ios::ate`	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
`ios::binary`	Binary mode. When a file is opened in binary mode, data are written to or read from it in pure binary format. (The default mode is text.)
`ios::in`	Input mode. Data will be read from the file. If the file does not exist, it will not be created and the open function will fail.
`ios::out`	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
`ios::trunc`	If the file already exists, its contents will be deleted (truncated). This is the default mode used by `ios::out`.

modes:

Enumerated Data Types

```
// each enumerator is assigned an integer starting from 0
enum Day
{
    MONDAY,     // 0
    TUESDAY,    // 1
    WEDNESDAY,  // 2
    THURSDAY,   // 3
    FRIDAY      // 4
};

// assign an enumerator to an integer
int x = THURSDAY;

// can not directly assign an int to an enum variable
Day day1 = static_cast<Day>(3);         // day1 = 3 is illegal!!
Day day2 = static_cast<Day>(day1 + 1);  // day2 = day1 + 1 is illegal!!

// compare enum values
bool b = FRIDAY > MONDAY // true because FIRDAY comes after MONDAY

// anonymous enum types can be used when you don't need to define variables
enum {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};

// can specify integer values for all or some enumerators
enum Color {RED, ORANGE, YELLOW = 9, GREEN, BLUE};
```

Strongly typed enumerators (`enum class`)

```
// can specify multiple enumerators with the same name, within the same scope
enum class President {MCKINLEY, ROOSEVELT, TAFT};
enum class VicePresident {ROOSEVELT, FAIRBANKS, SHERMAN};

// can not directly assign a strongly typed enum to an integer
// int x = President::MCKINLEY is illegal!!
int x = static_cast<int>(President::MCKINLEY);

// can specify any integer data type as the underlying type
enum class Day : char {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
```

Structs

```
// declare a struct
struct CityInfo
{
    string cityName;
    string state;
    long population;
    int distance;
};

// define a struct
CityInfo location;
CityInfo cities[20];

// initialize a struct
CityInfo location = {"Asheville", "NC", 50000, 28};
CityInfo location = {"Atlanta"};  // only cityName is initialized
CityInfo cities[2] = {{"Asheville", "NC", 50000, 28},
                      {"Atlanta", "GA", 45000, 90}};

// access struct members
location.population = 4750;
cout << location.population << endl;

// declare a nested struct
struct EmployeeInfo
{
    string name;
    int employeeNumber;
    CityInfo birthPlace;
};

// access nested struct members
EmployeeInfo manager;
manager.birthPlace.population = 4750;
cout << manager.birthPlace.population << endl;
```

Note: *By default, structures are passed to functions by value.*
Note: *You can return local structs defined in functions unlike arrays.*

Dynamically allocating structs

```
struct Circle
{
    double radius;
    double diameter;
    double area;
};

Circle* cirPtr = new Circle;
Circle* circles = new Circle[5];

// access members after dereferencing the struct pointer
cirPtr->radius = 1.1;
cirPtr->area = 2.2;

// array elements are structs, not pointers
for (int i = 0; i < 5; i++)
{
    circles[i].radius = 1.1 * i;
    circles[i].area = 2.2 * i;
}
```

Classes

Classes are usually made up of a specification file and an implementation file with extensions `.h` and `.cpp`; however it is also possible to put everyting inside a single `.h` file. As a simple example, see the specification and implementation files of the Rectangle class.

Copy constructors and destructors

See the ContactInfo class.

Objects

Here's some example usage of the Rectangle and ContactInfo classes:

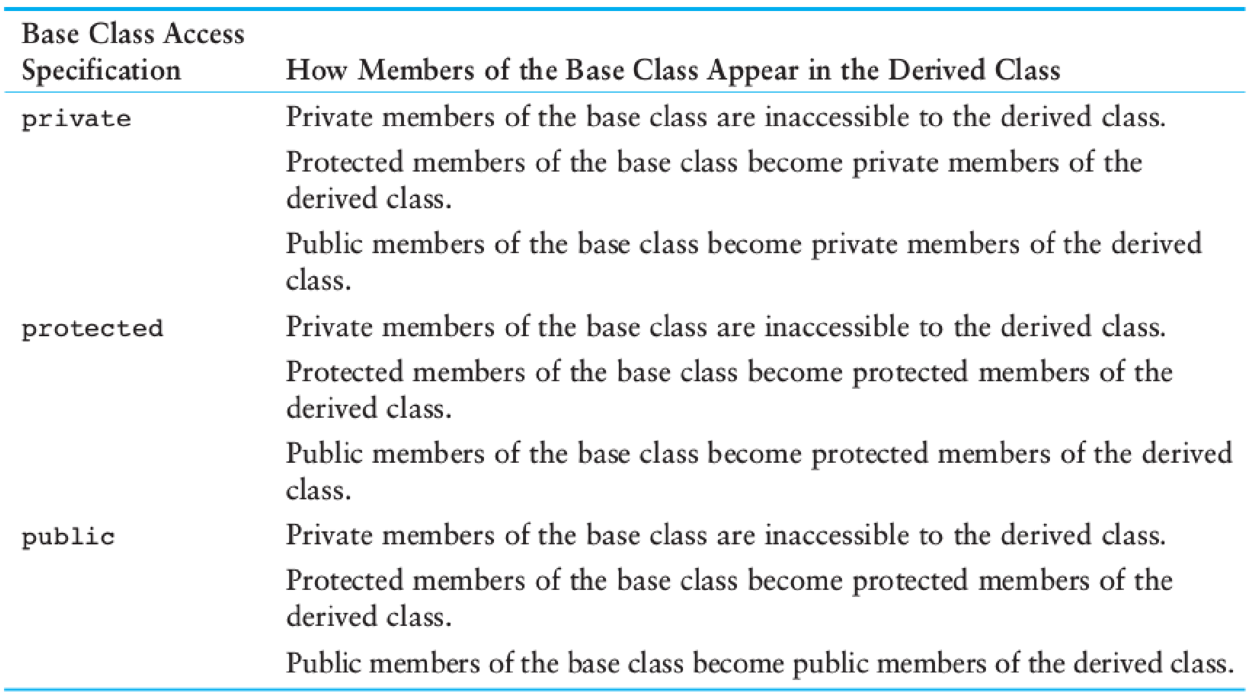
```
// define an object from the Rectangle class (lives on stack)
Rectangle box1(12.8, 9.4);

// initialize the new object with width & length of box1 (lives on stack)
// c++ automatically creates a default copy constructor if it's not defined by programmer
Rectangle box2 = box1;  // numObjects not incremented because it's not handled in default copy
constructor
box2.setWidth(4.7);     // assign a new width to box2

// call static member function
cout << "Number of objects: " << Rectangle::getNumObjects();    // Number of objects: 1

// define a pointer to a ContactInfo class object
// this object lives on the heap and should be deleted manually
ContactInfo* contactPtr = new ContactInfo("Kristen Lee", "555-2021");
contactPtr->getName();

// create a new contact and copy name & phone number from the previous one (lives on stack)
ContactInfo newContact = *contactPtr;       // our copy constructor is called here
automatically
const char* newName = newContact.getName(); // same as "Kristen Lee"

// the destructor is called here automatically
delete contactPtr;
contactPtr = nullptr;   // good practice for preventing errors

// no need to delete other objects because they live on stack and will get deleted when the
calling function returns
```

Inheritence

The parent class’s constructor is called before the child class’s constructor.
The destructors are called in reverse order, with the child class’s destructor being called first.

See the Cube class which inherits from the Rectangle class.

Base class access specification



Base Class Access Specification	How Members of the Base Class Appear in the Derived Class
`private`	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become private members of the derived class.
	Public members of the base class become private members of the derived class.
`protected`	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become protected members of the derived class.
`public`	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become public members of the derived class.

NOTE: If the base class access specification is left out of a declaration, the default access specification is `private`.

Polymorphism

Any class that has a virtual member function should also have a virtual destructor.
Even if the class doesn't normally require a destructor, it should still have an empty virtual destructor.

See the GradedActivity and PassFailActivity and then the example usage below:

```
PassFailActivity pfActivity(70);
pfActivity.setScore(72);
displayGrade(pfActivity);

// polymorphism requires pass by reference or by pointer
void displayGrade(const GradedActivity &activity)
{
    cout << "The activity's letter grade is " << activity.getLetterGrade() << endl;
}
```

Abstract Classes

A **pure virtual function** is a virtual member function of a base class that **must be overridden**.
When a class contains a pure virtual function as a member, that class becomes an **abstract class**.

```
// this is a pure virtual function
virtual void foo() = 0;
```

Exceptions

Throwing an Exception

```
double divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        // throws an exception of type "string"
        throw "ERROR: Cannot divide by zero.\n";
    }
    else
    {
        return static_cast<double>(numerator) / denominator;
    }
}
```

Handling an Exception

```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)  // only catches exceptions of type "string"
{
    cout << exceptionString;
}
```

There are two possible ways for a thrown exception to go uncaught:

1. The try/catch construct contains no catch blocks with an exception parameter of the right data type.
2. The exception is thrown from outside a try block.

In either case, the exception will cause the entire program to abort execution.

If an exception is thrown by the member function of a class object, then the class destructor is called. If any other objects had been created in the try block,their destructors will be called as well.

Multiple Exceptions

```
// define a custom exception class
class MyException: public std::exception
{
    public:
        MyException(const char* message) : msg(message) {}
        MyException(const std::string& message): msg(message) {}

        const char* what() const throw()
        {
            return msg.c_str();
        }

    protected:
        std::string msg;
};

double divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        // throws an exception of type "string"
        throw "ERROR: Cannot divide by zero.\n";
    }
    else if (numerator < 0 || denominator < 0)
    {
        // throw an exception of type "MyException"
        throw MyException("Negative arguments not allowed!");
    }
    else
    {
        return static_cast<double>(numerator) / denominator;
    }
}

// use the function
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (string msg)  // only catches exceptions of type "string"
{
    cout << msg;
}
catch (MyException& e)
{
    cout << "Error: " << e.what() << endl;
}
```

Rethrowing an Exception

```
try
{
    // assume that this function throws an exception of type "exception"
    doSomething();
}
catch(exception)
{
    throw;  // rethrow the exception
}
```

Standard Exceptions

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by new.
std::bad_cast	This can be thrown by dynamic_cast.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by typeid.

Exception	Description
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is use
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created.
std::out_of_range	This can be thrown by the 'at' method, e.g. a std::vector and std::bitset<
std::runtime_error	An exception that theoretically cannot be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occurred when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Templates

Function Templates

```
template <class T>
void swap(T &i, T &j)
{
    T temp = i;
    i = j;
    j = temp;
}

swap(a, b);
```