C++ Summary

Structures

```
struct name {
  type1 element1;
  type2 element2;
  ...
} object_name;     // instance of name
name variable;     // var. of type name
variable.element1; // ref. of element
name* varp;        // pointer to structure
varp->element1;  // member of structure
                    reached with a pointer
```

Console Input/Output

C++ console I/O

cout<< console out, printing to screen
cin>> console in, reading from keyboard
cerr<< console error
clog<< console log

```
cout<<"Please enter an integer: ";
cin>>i;
cout<<"num1: "<<i<<"\n"<<endl;
```

Control Characters

\b backspace **\f** form feed **\r** return
\' apostrophe **\n** newline **\t** tab
\nnn character # nnn (octal) **\"** quote
\NN character # NN (hexadecimal)

Functions

Passing Parameters by Value

```
function(int var);        // passed by value
```

Variable is passed into the function and can be changed, but changes are not passed back.

Passing Parameters by Constant Value

```
function(const int var);
```

Variable is passed into the function but cannot be changed inside the function.

Pass by Reference

```
function(int &var);      // pass by reference
```

Variable is passed into the function and can be changed, changes are passed back.

Pass by Constant Reference

```
function(const int &var);
```

Variable cannot be changed in the function.

Default Parameter Values

```
int add(int a, int b=2) {
  int r;     // b is 2, if
  r=a+b;     // no second parameter was given
  return r;
}
```

Overloading Functions

Functions can have the same name as long as the parameters are of different types. The return value cannot be the only difference.

```
// takes and returns integers
int divide (int a, int b) {
  return (a/b); }

// takes and returns floats
double divide (double a, double b) {
  return (a/b); }

divide(10,2); // returns 5
divide(10.0,3.0); // returns 3.33333333
```

Namespaces

```
namespace identifier {
   namespace-body;
}

namespace      first  { int var = 5; }
namespace second  { double var = 3.1416; }

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

using namespace allows for the current nesting level to use the appropriate namespace

```
using namespace identifier;

namespace      first  { int var = 5; }
namespace second  { double var = 3.1416; }

int main () {
  using namespace second;
  cout << var << endl;
  cout << (var*2) << endl;
  return 0;
}
```

Exceptions

```
try {
  // code to be tried...
  statements;            // if fails, exception is set
  throw exception;    // direct exception generation
}
catch ( type exception) {
  // code in case of exception
  statements;
}
catch(…) { }
```

Class Syntax

```
class classname {
public:
  classname( parms); // constructor
  ~ classname(); // destructor
  type  member1;
  type member2;
protected:
  type member3;
  ...
private:
  type member4;
} objectname;        // instance of classname

// constructor (initializes variables)
classname:: classname( parms) { }

// destructor (deletes variables)
classname::~ classname() { }
```

public members are accessible from anywhere where the class is visible
private members are only accessible from members of the same class or of a friend class
protected members are accessible from members of the same class, members of the derived classes and a friend class
constructors may be overloaded just like any other function. You can define two identical constructors with difference parameter lists

Class Example

```
class CSquare { // class declaration
public:
  void Init(float h, float w);
  float GetArea(); // functions
private:     // available only to CSquare
  float h,w;
} ;

void CSquare::Init(float hi, float wi){
  h = hi; w = wi; }

float CSquare::GetArea() {
  return (h*w); }

// example declaration and usage
CSquare theSquare;
theSquare.Init(8,5);
float area = theSquare.GetArea();

// or using a pointer to the class
CSquare *theSquare=new CSquare( );
theSquare->Init(8,5);
float area = theSquare->GetArea();
delete theSquare;
```

Inheritance

```
class Person{
            string name;
            int birthYear;
public:
   Person(string name, int birthYear)  {
      this->name=name;this->birthYear=birthYear;
    }
   void print() {cout<<name<<' '<<birthYear<<' ';}
   void setBirthYear(int birthYear){
      this->birthYear= birthYear;
   }
};

class Employee: public Person{
    int employmentYear;
public:
 Employee(string name, int birthYear,
    int employmentYear):Person(name, birthYear){
      this->employmentYear= employmentYear;
 }

  void print(){ Person::print();
     cout<<employmentYear<<endl;
  }
  void setEmploymentYear(int employmentYear){
        this->employmentYear=employmentYear;
   }
};

int main() {
            Person p("Garfield", 1965);
            p.print(); // Garfield 1965
            cout<<endl;
            Employee e("Ubul", 1964, 1965);
            e.print(); // Ubul 1964 1965
            return 0;
}
```

Visibility Of Members After Inheritance

	in base classs		
inheritance	**public**	**protected**	**private**
public	public	protected	-
protected	protected	protected	-
private	private	private	-

Advanced Class Syntax

Class TypeCasting

reinterpret_cast < newtype>(expression);
dynamic_cast < newtype>(expression);
static_cast < newtype>(expression);
const_cast < newtype>(expression);

Templates

Function templates

Definition of a function template:

```
template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b); // return the larger
}

void main () {
  int a=9, b=2, c;
  float x=5.3f, y=3.2f, z;
  c=GetMax(a,b);
  z=GetMax(x,y);
}
```

Class templates

```
template <class T>
class Pair {
  T x,y;
public:
  Pair(T a, T b) { x=a; y=b; }
  Pair(Pair<T>& p) { x=p.x; y=p.y; }
  T GetMax();
};
```

```
template <class T>
T Pair<T>::GetMax()
{
  T ret;
  ret = x>y?x:y; // return larger
  return ret;
}

int main () {
        Pair <int> theMax (80, 45);
        cout << theMax.GetMax();
        return 0;
}
```

How to create a class template from a class:

- Replace "**class** Pair" with "**template** <**class** T> **class** Pair" starting the class definition
- Replace the exact type with the template parameter name within class and method definitions, eg.: **int** → T
- For methods defined **outside the class definition** replace **int** Pair::GetMax() {} → **template** <**class** T> T Pair<T>::GetMax() {}
- Replace the class name with the class name decorated with the template parameters wherever the class is used as parameter or return value: Pair(**Pair**& p) { x=p.x; y=p.y; } → Pair(**Pair<T>**& p) { x=p.x; y=p.y; }

File I/O

File I/O is done from **fstream**, **ofstream**, and **ifstream** classes.

```
#include <fstream>   // read/write file
#include <ofstream> // write file
#include <ifstream>  // read file

using namespace std;
```

File Handles
A file must have a file handle (pointer to the file) to access the file.

```
ifstream infile; // create handle called infile
ofstream outfile; // a handle for writing
fstream f;        // handle for read/write
```

Opening Files
After declaring a file handle, the following syntax can be used to open the file

void open**(const char** * fname, **ios::**mode**)**;
fname should be a string, specifying an absolute or relative path, including filename. **ios::** mode can be any number of the following:

ate	Initial position: end of file
app	Output is appended at the end
trunk	Existing file is erased
binary	Binary mode
in	Reads (file must exist)
out	Empties and writes (creates file if it doesn't exist)
out | trunc	Empties and writes (creates file if it doesn't exist)
out | app	Appends (creates file if it doesn't exist)
in | out	Reads and writes; initial position is the beginning (file must exist)
in | out | trunc	Empties, reads, and writes (creates file if it doesn't exist)

```
ifstream f;    // open input file example
f.open("input.txt", ios::in);
ofstream f;  // open for writing in binary
f.open("out.txt", ios::out | ios::binary
| ios::app);
```

Closing a File
A file can be closed by calling the handle's close function f.**close()**;

Writing To a File (Text Mode)

The operator **<<** can be used to write to a file. Like **cout**, a stream can be opened to a device. For file writing, the device is not the console, it is the file. **cout** is replaced with the file handle.

```
ofstream f;                 // create file handle
f.open("output.txt")       // open file
f <<"Hello World\n"<<a<<b<<c<<endl;
```

Reading From a File (Text Mode)

The operator **>>** can be used to read from a file. It works similar to cin. Fields are separated in the file by spaces.

```
ifstream f("c:\\adat.txt"); // Open file

char c;
while(f>>c) // Read while not error
{
  cout<<c; // Process c
}
```

I/O State Flags

Flags are set if errors or other conditions occur. The following functions are members of the file object

handle.**bad()** returns true if a failure occurs in reading or writing
handle.**fail()** returns true for same cases as bad() plus if formatting errors occur
handle.**eof()** returns true if the end of the file reached when reading
handle.**good()** returns false if any of the above were true

Stream Pointers

handle.**tellg()** returns pointer to current location when reading a file
handle.**tellp()** returns pointer to current location when writing a file
to seek a position in reading a file:
handle.**seekg(** position**)**;
handle.**seekg(** offset, direction**)**;

to seek a position in writing a file:
handle.**seekp(** position**)**;
handle.**seekp(** offset, direction**)**;

direction can be one of the following:
ios::beg beginning of the stream
ios::cur current position of the stream pointer
ios::end end of the stream

Binary Files

buffer is a location to store the characters, **numbytes** is the number of bytes to written or read.

write(const char * buffer, numbytes**)**;
read(char * buffer, numbytes**)**;

Output Formatting

```
streamclass f;              // declare file handle
f.flags(ios_base:: flag)  // set output flags
```

possible flags:
dec fixed hex oct scientific internal left right uppercase boolalpha showbase showpoint showpos skipws unitbuf
adjustfield left | right | internal
basefield dec | oct | hex
floatfield scientific | fixed

f.**fill()** get fill character
f.**fill(**ch**)** set fill character ch
f.**precision(** numdigits**)** sets the precision for floating point numbers to numdigits
f.**put(** c**)** put a single char into output stream
f.**setf(** flag**)** sets a flag
f.**setf(** flag, mask**)** sets a flag w/value
f.**width()** returns the current number of characters to be written
f.**width(**n**)** sets the number of chars to be written

Dynamic Memory in C++

Allocate Memory

Syntax: pointer = **new** type [*size*];

```
int *ptr;                  // declare a pointer
ptr = new int;             // create a new instance
ptr = new int [5];         // new array of ints
```

Deallocate Memory

Syntax: **delete** pointer; or **delete**[] pointer;

```
delete ptr; // delete a single int
delete [] ptr // delete array
```

Class Reference

Friend Classes/Functions

```
class CSquare;          // declare CSquare
class CRectangle {
  int width, height;
public:
  void convert (CSquare a);
};

class CSquare {         // we want to use the
private:                // convert function in
  int side;             // the CSquare class, so
public:                 // use the friend keyword
  void set_side (int a) { side=a; }
  friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
  width = a.side; // access private member of
  height = a.side; // a friend class
}

CSquare sqr;
CRectangle rect;        // convert can be
sqr.set_side(4);        // used by the
rect.convert(sqr);      // rectangle class
```

Friend Classes/Functions

```
class CSquare;        // declare CSquare
class CRectangle {
  int width, height;
public:
  void convert (CSquare a);
};
```

The CSquare class with the **friend** keyword authorizes the CRectangle class and the Change global function to access its private and protected members:

```
class CSquare {
private:
  int side;
public:
  void set_side (int a) { side=a; }
  friend class CRectangle;
  friend void Change(CSquare s);
};

void CRectangle::convert (CSquare a) {
  width = a.side;  // the private  member "side" of
  height = a.side; // CSquare is accessed here
}

void Change(CSquare s)
{
  s.side += 5; // the private  member "side" of
               // CSquare is accessed here
}
```

Constructor calling order

1. Calling virtual base class constructor(s)
2. Calling direct, non-virtual base class constructor(s)
3. Constructing own parts
   a. Setting pointers to virtual base class parts
   b. Setting pointers of VFT
   c. Calling constructors of aggregated parts
4. User-defined parts of the constructors

Destructor calling order

1. User-defined parts of the destructor
2. Destructor(s) of aggregated components
3. Calling direct, non-virtual base class destructor(s)
4. Calling virtual base class destructor(s)