**Function Overloading:**

Overloading the function with different parameters

Code: <http://cpp.sh/6aac>

**Class and Object:**

Class is nothing but gathering all the attributes under the one data type.

Class Human

{

Public: Known to everyone even outside of the class

Private: Only known with in the class

Protected: known with in the class as well as derived class/inherited class/sub class

};

Object is nothing but a variable of this class in the real word.

Human Vasu;

Code: <http://cpp.sh/2lph>

**Class Variables and methods:**

Code: <http://cpp.sh/7rsj>

**Print Hex Oct Dec:**

std::cout << std::hex << x << std::endl;

**‘NEW’ Keyword:**

If we want to create an object to the class dynamically, we use NEW keyword. It will return the address of that class.

We can use the arrow operator to access the class variable or methods.

* Object is created in the HEAP since we do are not sure how much memory it will take.

Code: http://cpp.sh/2gm4

**Define class methods:**

Instead of having both definition and declaration in the same class, we declare the class methods outside the class using scope resolution operation (::).

Code: http://cpp.sh/2gwf

**Initialization of class variable:**

Class variables can be initialized with in the class if they are not *STATIC* type.

If the class variables are *STATIC* type we have to declare the variables outside the class.

Code: <http://cpp.sh/9syx>

**Access specifier:**

Access Specifiers are PUBLIC, PRIVATE and PROTECTED

PUBLIC: We can access these variable and methods everywhere in the program just by creating the object to the class.

Code: <http://cpp.sh/7inc>

PRIVATE: Hide the class variable from outside world. We can set/initialize the private variable or we call the private methods by using the public methods or public variables.

Code: <http://cpp.sh/9bky>

PROTECTED: We can access the Class protected methods in both within the class and derived class. It will hide to access from outside world.

Code: http://cpp.sh/5vue5

**Constructor:**

Coed: <http://cpp.sh/7zxk>’

**Over loaded Constructor:**

Calling the constructor, passing different data type to the function

For every object one constructor will be called either default or with parameters

Code: <http://cpp.sh/34ho>

**Destructor:**

Destructor is nothing but destroy the object which was created recently to the class using ‘tild’ operator.

**Virtual Function:**

Virtual int X () { }

The member function X has been declared as virtual in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of X in the example above, all calls would return from base class. Because in all cases, the version of the base class would have been called instead.

**Abstract Base Class:**

Virtual int X ()= 0; “Pure virtual function”

Abstract base classes are something very similar to the base class.

They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition.

Classes that contain at least one pure virtual function are known as abstract base classes.

**Example for Inheritance with Virtual Function**

<http://cpp.sh/95hu>

**http://cpp.sh/8wth**

More: [**http://cpp.sh/8xw**](http://cpp.sh/8xw)

**Static Variables:**

Const mean does not change the value

Static mean does not move the place

**http://cpp.sh/7aeg**

**Friend Function**

A **Friend Function** is a function that can have access to even the private, protected members of a class. This can be an ordinary function or a member of another class. It acts as a bridge between two classes.

http://cpp.sh/52ru

**Template:**

Template is used to call the same methods with different data types!

Code: <http://cpp.sh/8m7jr>

**Vector:**

<http://cpp.sh/5ff4>

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**

**Vector Insert:** [**http://cpp.sh/4d26**](http://cpp.sh/4d26)

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them.

**Data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

[www.linkedin.com/in/srinivasaraosakhamuri](http://www.linkedin.com/in/srinivasaraosakhamuri)

ASSERT\_\* versions generate fatal failures when they fail, and **abort the current function**. EXPECT\_\* versions generate nonfatal failures, which don't abort the current function. Usually EXPECT\_\* are preferred, as they allow more than one failures to be reported in a test. However, you should use ASSERT\_\* if it doesn't make sense to continue when the assertion in question fails.

|  |  |  |
| --- | --- | --- |
| **Fatal assertion** | **Nonfatal assertion** | **Verifies** |
| ASSERT\_TRUE(*condition*); | EXPECT\_TRUE(*condition*); | *condition* is true |
| ASSERT\_FALSE(*condition*); | EXPECT\_FALSE(*condition*); | *condition* is false |

<https://code.google.com/p/googletest/wiki/Primer>

++i increments i and evaluates to the new value of i.

i++ evaluates to the old value of i, and increments i

**Serial** is an umbrella word for all that is "Time Division Multiplexed", to use an expensive term. It means that the data is sent spread over time, most often one single bit after another. All the protocols you're naming are serial protocols.

**UART**, for Universal Asynchronous Receiver Transmitter, is one of the most used serial protocols. It's almost as old as I am, and very simple. Most controllers have a hardware UART on board. It uses a single data line for transmitting and one for receiving data. Most often 8-bit data is transferred, as follows: 1 start bit(low level), 8 data bits and 1 stop bit(high level). The low level start bit and high level stop bit mean that there's always a high to low transition to start the communication. That's what describes UART. No voltage level, so you can have it at 3.3 V or 5 V, whichever your microcontroller uses. Note that the microcontrollers which want to communicate via UART have to agree on the transmission speed, the bit-rate, as they only have the start bit's falling edge to synchronize. That's called asynchronous communication.

For long distance communication (That doesn't have to be hundreds of meters) the 5 V UART is not very reliable, that's why it's converted to a higher voltage, typically +12 V for a "0" and -12 V for a "1". The data format remains the same. Then you have **RS-232** (which you actually should call EIA-232, but nobody does.)

The timing dependency is one of the big drawbacks of UART, and the solution is **USART**, for Universal Synchronous/Asynchronous Receiver Transmitter. This can do UART, but also a synchronous protocol. In synchronous there's not only data, but also a clock transmitted. With each bit a clock pulse tells the receiver it should latch that bit. Synchronous protocols either need a higher bandwidth, like in the case of Manchester encoding, or an extra wire for the clock, like SPI and I2C.

**SPI** (Serial Peripheral Interface) is another very simple serial protocol. A master sends a clock signal, and upon each clock pulse it shifts one bit out to the slave, and one bit in, coming from the slave. Signal names are therefore SCK for clock, MOSI for Master Out Slave In, and MISO for Master In Slave Out. By using SS (Slave Select) signals the master can control more than 1 slave on the bus. There are two ways to connect multiple slave devices to one master, one is mentioned above i.e. using slave select, and other is daisy chaining, it uses less hardware pins(select lines), but software gets complicated.

**I2C** (Inter-Integrated Circuit, pronounced "I squared C") is also a synchronous protocol, and it's the first we see which has some "intelligence" in it; the other ones dumbly shifted bits in and out, and that was that. I2C uses only 2 wires, one for the clock (SCL) and one for the data (SDA). That means that master and slave send data over the same wire, again controlled by the master who creates the clock signal. I2C doesn't use separate Slave Selects to select a particular device, but has addressing. The first byte sent by the mast er holds a 7 bit address (so that you can use 127 devices on the bus) and a read/write bit, indicating whether the next byte(s) will also come from the master of should come from the slave. After each byte receiver must send a "0" to acknowledge the reception of the byte, which the master latches with a 9th clock pulse. If the master wants to write a byte the same process repeats: the master puts bit after bit on the bus and each time gives a clock pulse to signal that the data is ready to be read. If the master wants to receive data it only generates the clock pulses. The slave has to take care that the next bit is ready when the clock pulse is given. This protocol is patented by NXP(formerly Phillips), to save licensing cost, Atmel using the word TWI(2-wire interface) which exactly same as I2C, so any AVR device will not have I2C but it will have TWI.

Two or more signals on the same wire may cause conflicts, and you would have a problem if one device sends a "1" while the other sends a "0". Therefore the bus is wired-OR'd: two resistors pull the bus to a high level, and the devices only send low levels. If they want to send a high level they simply release the bus.

**TTL** (Transistor Transistor Logic) is not a protocol. It's an older technology for digital logic, but the name is often used to refer to the 5 V supply voltage, often incorrectly referring to what should be called UART.

* **bit** A single binary digit, that can have either value 0 or 1.
* **byte** 8 bits.
* **nybble** 4 bits.
* **word** 32 bits
* **halfword** 16 bits
* **doubleword** 64 bits