## Overview

Object-oriented programming is a programming style that captures the behavior of the real world in a way that hides detailed implementation, and it allows the problem solver to think in terms of problem domain. C++ was created with two goals: to make it compatible with ordinary C, and to extend C with OO constructs.

FAQ:http://www.parashift.com/c++-faq-lite/

Object-oriented programming is a data-centered view of programming, in which data and behavior are strongly linked. Data and behavior are conceived of as classes whose instances are objects. Objects are class variables, and object-oriented programming allows abstract data structures to be easily created and used.

A C++ program is a collection of declarations and functions that begin executing with the function `main()`. The program is compiled with a first phase executing any preprocessor directives, such as

```
#include <string>
```

Namespaces were introduced to provide a scope that allows different code providers to avoid global name clashes. The

```
namespace std;
```

is reserved for use with the standard libraries. The `using` declaration allows the identifiers found in the standard library to be used without being qualified. Without this declaration the program would have to use `std::string`.

In OO terminology, a variable is called an *object*. A constructor is a member function whose job is to initialize an object of its class. Constructors are invoked whenever an object of its class is created. A destructor is a member function whose job is to finalize a variable of its class. The destructor is called implicitly when an automatic object goes out of scope.

## Native types

The simple native types in C++ are `bool`, `double`, `int` and `char`. These types have a set of values and representation that is tied to the underlying machine architecture on which the compiler is running.
C++ simple types can be modified by the keywords `short`, `long`, `signed` and `unsigned` to yield further simple types.

| Fundamental data types | | |
|---|---|---|
| bool | | |
| char | signed char | unsigned char |
| wchar_t | | |
| short | int | long |
| unsigned short | unsigned | unsigned long |
| float | double | long double |

A variable declaration associates a type with a variable name. A declaration of a variable constitutes a definition, if storage is allocated for it. The definition can be thought as a creating the object.

```
int i;
```

A definition can also initialize the value of the variable

```
int i = 0;
int i = 0,j;
```

C++ declarations are themselves statements and they can occur throughout a block.

An automatic type conversion can occur across assignments and mixed statements. A promotion (widening) is usually well behaved, but demotions (narrowing) can lose information. In mixed statements the type conversion follows promotion rules.

In addition to implicit conversions, there are explicit conversions called `casts`.

```
static_cast<double>(i)
const_cast<double>(constVar)
```

Older C++ systems allow an unrestricted form of cast with

```
(type)expression or type(expression)
```

## Expressions

The arithmetic operators in C++ are +,-,*,/,%. To raise to a power one can use the function pow. Arithmetic expressions are consistent with expected practise, with one important difference: division operator. The result of the division operator depends on its arguments

```
a = 3 / 2;
a = 3 / 2.0;
```

Relational (<, >, <=, >=), equality (==,!=) and logical (!,&&,||) operators work as expected. In the evaluation of expressions that are the operands of && and ||, the evaluation process stops as soon as the outcome is known. If expr1 is false, then in

```
expr1 && expr2
```

expr2 will not be evaluated. Similarly, if expr1 is true, then in

```
expr1 || expr2
```

expr2 will not be evaluated since the value of the logical expression is already determined.

The conditional operator ?: is a ternary operator, taking three expressions as operands. In a construct such as

```
expr1 ? expr2 : expr3
```

expr1 is evaluated first. If it is true, expr2 is evaluated, and that is the value of the conditional expression as a whole. If expr1 is false then expr3 is evaluated. For example

```
x = (y < z) ? y : z;
```

assigns the smaller of two values to x.

C++ provides also assignment operators that combine an assingment and some operator like

```
a += b; // a = a + b
a *= a + b; // a = a*(a+b)
i++; // i = i + 1
i- -; // i = i - 1
j = i++; // j = i; i = i + 1
j = ++i; // i = i + 1; j = i
```

Notice that the two last examples are not equivalent.

## Statements

The general forms of basic control flow statements are

```
if ( condition ) statement
if ( condition ) statement1 else statement2
while ( condition ) statement
for (init; condition; increment) statement
do statement while ( condition )
switch ( condition ) statement
```

To interrupt the normal flow of control within a loop, two special statements can be used: `break` and `continue`

Example: a simple for loop

```
for(int i = 0; i < 3; i++){
  cout << i << endl;
}
```

Example: a while loop

```
while (getline(file,line)){
  cout << line.length() << endl;
}
```

## Functions

A C++ program is made of one or more functions, one of which is `main()`. The form for a function definition is

```
type name(parameter-list) {
  statements
}
```

The type specification that precedes the function name is the return type, and the value is returned with statement `return`. If the function does not return any value, the return type of the function is `void`.

The parameters in the parameter list can be given default arguments

```
void myFunction(int i, int j = 1){...}
```

It is possible to overload functions

```
void myFunction(int i){...}
void myFunction(double d){...}
```

## Pointers

Pointers are used to reference variables and machine addresses. They can be used to access memory and manipulate addresses. Pointer variables can be declared in programs and then used to take addresses as values. The declaration

```
int* p;
```

declares p to be of type pointer to int. The legal range of values for any pointer always includes the special address 0.

```
int* p = &i;
```

The variable p here can be thought of as referring to i or pointing to i or containing the address of i.

The dereferencing or indication operator * can be used to return the value of the variable the pointer points to

```
int j = *p;
```

Here *p returns the value of variable i.

## Arrays

An array declaration is of the form

```
int a[size];
```

The values of an array can be accessed by a[expr], where expr is an integer expression getting values from 0 to (size-1). Arrays can be initialized by a comma separated list

```
int a[3] = {1, 2, 3};
```

or element by element a[0] = 1; a[1] = 2; a[2] = 3;. Arrays can be thought of constant pointers. Pointer arithmetic provides an alternative to array indexing

```
int *p = a;
int *p = &a[0];
```

Arrays can be of any type, including arrays of arrays. Multidimensional arrays can be formed by adding bracket pairs

```
int a[2][2]; // Not a[2,2]!
int b[2][5][200];
```

## Abstract Data Structures

ADT implementations are used for user-defined data types. A large part of the OO program design process involves thinking up the appropriate data structures for a problem.

The `structure` type allows the programmer to aggregate components into a single named variable. A structure has components, called members, that are individually named. Since the members of a structure can be of various types, the programmer can create aggregates suitable for describing complicated data.

Example: complex numbers

```
struct complex {
  double re,im;
};

complex c;
c.re = 1; c.im = 2;
```

## Member functions

The concept of `struct` is augmented in C++ to allow functions to be members. The function declaration is included in the structure declaration.

```
struct complex {
  void reset(){
    re = 0;
    im = 0;
  }
  double re,im;
};
```

The member functions can be `public` or `private`. Public members are visible outside the data structure, while private can be accessed only within the structure. Very often the data is hidden, and accessed only by functions built for that purpose.

Hiding data allows more easily debugged and maintained code because errors and modifications are localized.

## Classes

Classes are introduced by a keyword `class`. They are a form of struct whose default privacy specification is `private`. Thus struct and class can be used interchangeably with the appropriate access specifications.

```
class complex {
 public:
   void reset(){
     re = 0;
     im = 0;
   }
 private:
   double re,im;
};
```

Class adds a new set of scope rules to those of the kernel language. The scope resolution operator :: comes in two forms:

```
::i // refers to external scope
foo_bar::i // refers to class scope
```

Classes can nest. This means that there can be classes within classes. Referencing the outer or inner variables is done with the scope resolution operator.

```
class X {
  int i; // X::i
  public:
  class Y {
    int i; // X::Y::i
  };
};
```

The definition of a purely local class is unavailable outside their local scope.

Static variables are shared by all variables of that class and stored in only one place.

They keyword `this` denotes an implicitly declared self-referential pointer.

```
class X {
  public:
    X clone(){return (*this);}
};
```

The member functions can also be `static` and `const`. A static member function can exist independent of any specific variables of the class type being declared. Const member functions cannot change the values of the data members. Writing out const member functions and const parameter declarations is called const-correctness. It makes the code more robust.

```
class X {
  static int Y();
  int Z() const;
};
int X::Y(){}
int X::Z() const {}
```

## Input/Output

Output is inserted into an object of type `ostream`, declared in the header file `iostream.h`. An operator << is overloaded in this class to perform output conversions from standard types.

```
std::cout << "Hello world!" << std::endl;
```

Input is inserted in `istream`, and operator >> is overloaded to perform input conversions to standard types.

```
std::cin >> i;
```

Here "`using namespace std;`" would allow using these commands without the scope resolution operator.

File i/o is handled by including `fstream.h`. To read a variable $var$ from a file first open the file:

```
ifstream inFile("filename",ios::in);
inFile >> var;
```

A while loop can be made to read until EOF is reached: `while(!inFile.eof()){}`

It is also possible to read a whole line using `getline(ifstream,string)`.

Example: MET from high pt objects

File SimpleMET.h:

```
#ifndef SIMPLEMET_H
#define SIMPLEMET_H

#include "MyEvent.h"

class SimpleMET {
   public:
     SimpleMET();
     ~SimpleMET();

     void Add(MyTrack);
     void Add(MyJet);

     MyMET GetMET();
     double Value();

   private:
     MyMET etmiss;
};
#endif
```

File SimpleMET.cc:

```
#include "SimpleMET.h"

SimpleMET::SimpleMET(){
     etmiss.x = 0;
     etmiss.y = 0;
}
SimpleMET::~SimpleMET(){}

void SimpleMET::Add(MyTrack track){
     etmiss.x -= track.Px();
     etmiss.y -= track.Py();
}

void SimpleMET::Add(MyJet jet){
     etmiss.x -= jet.Px();
     etmiss.y -= jet.Py();
}

MyMET SimpleMET::GetMET(){
     return etmiss;
}

double SimpleMET::Value(){

     return etmiss.Value();
}
```