Lexical Conventions
A C++ preprocessor processes the source file before the C++ compiler compiles it. This allows rudimentary macros and including other files in the compilation. Only after the preprocessor finishes with the file does the compiler see it. The compiler only sees the tokens generated by the preprocessor. This preprocessor is slightly different from the C preprocessor.

Preprocessing steps
Source file characters are mapped to C++ character set newline characters

\uXXXX coding

<newline> is deleted (the file must end in a <newline>)

The file is tokenized - comments replaced with a space comments

// /* */

alternative tokens

<% { and && and_eq &= %>
%> } bitor | or_eq |=
:< [ or || xor_eq ^= :
>: ] xor ^ not !
%: # compl ~ not_eq !=
%:%: ## bitand &

Preprocessing directives are executed

Escape sequences and \uXXXX replaced by characters

Adjacent ordinary string literals are concatenated

The file is now tokenized and ready to be compiled

Preprocessor Directives

#if constant-expression
#endif identifier
#endif constant-expression
#endif
#endif

#include <xxx>

searches for "library" files

#include "xxx"

searches for user files then library files

#define identifier replacement
#define identifier ( args ) replacement
#define identifier replacement

in the definition

#pragma -> "param"
## -> causes concatenation of tokens before continuing

#undef identifier

undefines a macro defined with #define

#define identifier replacement

in the definition

#pragma -> "param"
## -> causes concatenation of tokens before continuing

#error message

forces an error with the indicated message

implementation-specific directives

# no effect

Predefined macros

_LINE_

the current line number

_FILE_

the current file name

_DATE_

the current date in the format "Mmm dd yyyy"

_TIME_

the current time in the format "hh:mm:ss"

_STDCC_

implementation dependent

#pragma cplusplus

199711L (version of the standard)

Keywords

reserved

asm auto bool break case catch char class const
cast continue default delete do double
dynamic_cast else enum explicit export extern
false float for friend goto if inline int long
mutable namespace new operator private protected
public register reinterpret_cast return short
signed sizeof static static_cast struct switch
template this throw true try typedef typeid
volatile wchar_t while

alternative representations

and and_eq bitand bitor compl not not_eq or or_eq
xor xor_eq

Operators and punctuators

{ } [ ] :: . .* + - * / % ^ & | ~ != < > += -=
*+= /= %= ^= &= |= << >>= <<= == != <= >= && ||
++ -- , ->* -> and and_eq bitand bitor compl not not_eq or or_eq
xor xor_eq

Escape Sequences

new-line NL(LF) \n
horizontal tab HT \t
vertical tab VT \v
backspace BS \b
carriage return CR \r
form feed FF \f
alert BEL \a
backslash \\ \\
question mark ? \?single quote ' \

double quote " \"
octal number ooo \ooo
hex number hhh \xhhhh
The C++ Language

Compilation units

A C++ program is made up of individually compiled "compilation units"

Compilation units communicate with each other through external symbols

Compilation units cannot see symbols in other compilation units that are not external

main function

There must be one top-level function called "main" with one of the following signatures:

```cpp
int main( ...arbitrary... ) {}  
int main( ... ) {}  
int main( int argc, char *argv[ ] ) {}  
```

`argv[0]` is the name of the program

`argv[1]` to `argv[n]` are the command line arguments

`argv[n+1] = 0`

`argc = n+1`

Types - simple types

Basic types

void

integer types

bool
char

- unsigned char
- signed char

short [int]

- unsigned short [int]
- signed short [int]

int

- unsigned int = unsigned
- signed [int] = int

long [int]

- unsigned long [int]
- signed long [int]
wchar_t

(bit fields)

floating point types

float
double

long double

Types that refer to other types

references

- references are variables that refer to other variables
- Can refer to element of an array.

references are initialized to be a synonym for another variable - they are not assigned to - what they refer to is assigned to

pointers

pointers to nonstatic class members

const

"read only"

volatile

"do not optimize"

Compound types

arrays

functions

struct, class

union

enum

anonymous enum

Strings

String literals exist and construct "array of char" objects with a trailing null character.

There are strings and wide strings

Do not confuse strings with the library std::string class

lvalues and rvalues

Originally lvalues were values that were allowed on the left side of an assignment and rvalues were not lvalues

lvalues are generally variables, references, pointer indirections (*x)

rvalues are generally constants, functions, non-reference function returns
Expressions

literals
  true, false, number, "x", 'c', 0
this
 ( expression )
identifier
  classname :: member
  namespacename :: member
  :: name (global)
  ::x::y \template{xxx}\ (as prefix)
operator @
  - <className>
  <conversion-function-id>

a[] array reference
f(x, x) function call
T(x, x) constructor
int(x) explicit conversion
typename x::y z()
x.y member access
x->y member access through pointer ( similar to (*x).y )
x++ and x--
  while('p++ = "q++" ;
typeid( exp ) or typeid( type-id )
  runtime type information
C style casts - do not use!!!
(int) x
C++ casts

  const_cast<TypeName> ( expression )
  dynamic_cast<TypeName> ( expression )
  reinterpret_cast<TypeName> ( expression )
  static_cast<TypeName> ( expression )

  const_cast
    cast away const
dynamic_cast
    casts pointer or reference to "superclass"
    similar to java casts on object pointers
reinterpret_cast
  "just reinterpret the bits"
static_cast
  does normal conversions " TypeName t(expression)"
++x and --x
  *x
  +x and -x
  &x (get the address of)
  !(x (boolean not)
  ~x (bit not)
sizeof( x ) and sizeof( T )
new T ( init args )
new T[6]
new ( placement args ) T ( init args )
new ( type-id ) ( init args )
new ( placement args ) ( type-id ) ( init args )
delete x and delete[] x
x,y
x->y
x*y and x/y and x%y
x+y and x-y
x<<i and x>>i (bit shift and stream insert and extract)
x < y and x >= y and x <= y and x >= y
x == y and x != y
x & y (bit and)
x ^ y (bit exclusive or)
x | y (bit or)
x && y (logical and)
x || y (logical or)
x ? : y : z
x=y x*=y x%=y x+=y x-=y x>>=y x<<=y x&=y xy ^=y
x|=y
throw exp;
x, y (comma sequencing)

C++ Operators

The operators are listed here in the order of their precedence from high to low
Parentheses may be used to group expressions in another way then precedence or associativity would do it by default
As only few people are familiar with the whole list, it might enhance readability to use parentheses in other cases as well or to format the program text for complex expressions according to the precedence of operators: Break lines at operators with lower precedence
As far as operators that are shared, the order of precedence is equivalent to that of ISO C

C++ Operators

postfix operators - left to right
++ -- -> . [] ()

  unary operators - right to left
  + -

  multiplicative operators - left to right
  * / %

  additive operators - left to right
  + -

  shift operators - left to right
  << >>

  relational operators - left to right
  < > <= >=

  equality operators - left to right
  == !=

  bitwise AND - left to right
  &

  bitwise exclusive OR - left to right
  ^

  bitwise inclusive OR - left to right
  |

  logical AND - left to right
  &&

  logical OR - left to right
  ||

  conditional operator - right to left
?:

  assignment operators - right to left
  *= /= %= += -= <<= >>= &= ^= |=

  comma operator - left to right ,
Standard Conversions
A conversion consists of a standard conversion, one user conversion, another standard conversion
at most one of
- lvalue to rvalue
- array to pointer
- function to pointer
at most one of
- integral promotion
  - char -> (unsigned) int
  - wchar_t or enum -> (unsigned) int or (unsigned) long int
  - bit-field -> (unsigned) int
  - bool -> int
- floating point conversion
  - float -> double
- integral conversion
  - bool to int (zero is false, else true)
  - int type -> int type (unsigned is modulo else undefined)
- floating point conversion
  - long double -> double -> float
- floating-integral conversion
  - floating -> integral (truncating)
- pointer conversion
  - 0 -> pointer
  - pointer to T -> pointer to void
  - pointer to class -> pointer to base
- pointer to member conversion
  - 0 -> pointer to member
  - pointer to member of Base -> pointer to member of Derived
- boolean conversion
at most one qualification conversion (const/volatile)

Statements
label: S;
; (the empty statement)
expression; (expression statement)
switch(int i = 5) S;
switch (int i = 5) {
  case 5: S;
  break;
  default: S;
}
{ S; S; }
if (b) S;
if (b) S; else S;
while(b) S;
  if, switch, and while conditions can be a declaration whose variable value is the value of the condition
do S; while(b);
for(int i=3, j=2; i < 5; i++) S;
break;
continue;
return x;
goto label;
try {S; S;} catch(T e) {S; S; }

Exceptions
try {} // can also surround function initializer
catch(Exception &e) {}
catch(...) {}
throw exp;
void f() throw int;
in a catch clause, throw with no operand rethrows the same exception
special functions
  void terminate() - called if no handler found
  void unexpected() - exception not listed in specification
  bool uncaught_exception() - true if processing an exception

Declarations
typedef IntPointer * INT
enum e { x, y = 3, z }
namespace name { ...
namespace { ...
namespace x = y::z;
using B::f;
using namespace X::Y::Z;
asm { "xxxxx" };
extern "C" { ...
extern "C" int f() { return 0; }

Declarators
const volatile int i, &j = i, *k;
int *f(int i) const, a[10];
int X::* pmi = &X::a;
int f( int i=1, ... ) const { return i+1; }
The ... argument stands for for varying number of arguments
argument i=1 defines a default value
S::S( int i=1 ) const : j(1), k(2) {}
S s(s1);
S ss=s1;

Function declarations
Method declarations
int g( int& i );
Operator declarations
int& operator++();
Modifiers

auto
stack allocated - automatic storage allocation
const
read only, internal linkage if not explicitly declared extern or previously declared to have external linkage
explicit
no implicit conversions allowed with this constructor
export
used for exporting template definition
extern
external linkage
friend
declared in the class granting friendship
inline
procedure expanded without function call
mutable
allow writing even in const class instance
register
allocate in a register - automatic storage allocation
static
exactly one - not a field in a class instance, internal linkage
virtual
method can be overridden
volatile
don’t optimize

Declarations and Definitions

Declarations
Declarations specify the type of an identifier but do not specify its value
Definitions
Definitions specify the type and value of an identifier
It is a definition unless function without body
extern specifier or extern “xxx” specifier (linkage specification) without initializer or function body
static data member in a class declaration
a class name declaration
typedef declaration
using declaration
using MySpace::f;
using directive (not allowed at class scope)
using MySpace;
language “functions”
delete new sizeof typeid
library functions
various functions and classes to interface to the operating system and support some language features

Constructors

Constructors are like functions but have the same name as the class
Constructors do not indicate a return type
class Point {
    int x, y;
public:
    Point( int px, int py);
}
Point::Point(int px, int py) : x(px), y(py) {}
Special operators

operator@ as a member function
operator@ as a non-member function

@a
(a).operator@ ()
operator@ (a)

a@b
(a).operator@ (b)
operator@ (a, b)

a=b
(a).operator= (b)
= must be a nonstatic member function

a[b]
(a).operator[] (b)
must be a nonstatic member function

a(b)
(a).operator() (b)
must be a nonstatic member function

a->
(a).operator-> ()
-> must be a nonstatic member function
the result returned by the operator is used as the lhs of a - operator call with the same rhs

a@
(a).operator@ (0)
operator@ (a, 0)

Rules for new and delete
new(x, y) X(z, w)
member function operator TYPE() is a conversion operator
defines conversion from base type to TYPE
syntax like a constructor
You can make default operators of a class inaccessible by declaring them private (they are already defined)

Variable Initialization

copy-initialization
S s = xxx;
parameter passing
function return
exception throwing
exception handling
{ initializers } for arrays and pod structures
direct initialization
new expressions
static_cast
functional type conversions
base and member initializations

Classes

Classes define the characteristics of objects

Classes can be
at top-level
in a namespace
nested in a class
name local in enclosing class - declarations can use
type names
static members
enumerators
local to a block
can use only from enclosing scope - declarations can use
type names
static variables

extern variables and functions
enumerators
cannot have static data members

Members of classes

Members can be
classes
types
functions
virtual functions can be overridden
pure virtual functions indicate that the class is an abstract
class and cannot be instantiated
conversion functions - operator X() {} (where X is a type)
data members (attributes)
bit-fields
int i: 10, : 2, : j: 3;

Members
cannot be extern, auto, or register
virtual functions can be overridden
overriding function is also virtual even if it is not declared virtual

Special members of Classes
$ indicates a default member will be implicitly declared if one is not declared

Constructors
$default constructor
C() {
}$copy constructor
C( C &c ) : c(c) {
coercing constructor
must use "explicit" to prevent automatic conversions
C( int i ) { "initialize this from i " }

Functions
$destructor - written ~ClassName()
$operator = (copy assignment operator)

Operators
* -> &
the this pointer
pod (plain old data) structs are special

Access to members of classes

Access defaults
classes default members to private
structs default members to public
unions default members to public

Access
access for members are defined by regions in the class declaration after the access keyword
public:
anyone can access
protected:
only used by members and friends of this class and derived classes
private:
only used by members and friends of this class
friends
friend class B;
friend void f();
a class or function granted access to private variables
nested classes are not automatically granted access
friends of friends are not friends
derived classes are not friends
Unions

Unions are like structures or classes except all of its fields lie on top of one another

```c
union U { int i; float f; }
```

The size of a union is the size of its largest member

This can save space

This can be used to convert from one representation to another

Unions are unsafe

Unions

can have member functions

cannot have virtual functions

cannot have a base class or be used as a base class

cannot have static members

cannot contain a class with

- non-trivial constructor
- non-trivial copy constructor
- non-trivial destructor
- non-trivial copy assignment operator

Anonymous unions

```c
union { int i; double d; }
```

no nested types or functions

all members declared in the same scope as the union anonymous unions in a namespace shall be declared static cannot have protected or private members

Class Definition Syntax:

class X : A, public virtual B { };

virtual means that only one instance of the base will be created even though there are more than one routes to a virtual base class through multiple inheritance nonvirtual base classes are always instantiated

the access control indicates how the base class members appear in the derived class. private base class members cannot be accessed this way

One definition rule

There can be only one definition for variables functions class type enumeration type templated functions and classes

Except for in a different translation unit class type enumeration type inline function with external linkage class template non-static template function static data member of a class template member function template template specialization with some template parameters not specified

Scopes

namespace scope

anything in a namespace definition

the top-level scope is a namespace scope
class scope

this is the scope where members are declared or defined

function prototype scope

function scope (for labels)

this is the body of a function definition

local scope

function try-block

block

catch phrase

Scopes

top-level - can contain

namespace
class
function

namespace - can contain

namespace
class
function
class - can contain

function
class
function - can contain

local

local - can contain
class
local

Library functions

types <stddef>

macros

NULL

offsetof(type, member-designator)
types

ptrdiff_t

size_t

implementation properties <limits> <climits> <cfloat>

numeric limits gives properties of numeric types

start and termination <cstdarg>

abort(), atexit( f ), exit( status )
dynamic memory management <new>

new and delete functions, class bad_alloc,

set_new_handler( f )
type identification <typeinfo>

class for describing runtime types (returned by typeid operator)

exception handling <exception>

classes exception and bad_exception

set_unexpected( f )

set_terminate( f )

uncaught_exception() // true if processing exception

other runtime support

<cstdarg> for variable-length argument lists

<csetjmp> for setjmp -- do not use

<ctime> for timing etc.

<csignal> interface to signals

<cstdlib> runtime environment
Libraries
Diagnostics library
  <stdexcept> Exception classes
  <cassert> Assertions
  <cerrno> Error numbers
General utilities library
  <utility> Utility components
  <functional> Function objects
  <memory> Memory
  <ctime> Date and time
Strings library
  <string> Character traits and String classes
  <ctype> <cwctype> <cstring> <cwchar> <cstdlib> Null-terminated sequence utilities and character identification functions
Localization library
  <locale> Locales and Standard locale categories
  <clocale> C library locales
Numerics, Collections, Algorithms, Streams and IO
Containers library
  <deque> <list> <stack> <vector> Sequences
  <map> <set> Associative containers
  <bitset> bitset
Iterators library
  <iterator> Iterator primitives, predefined iterators, stream iterators
Algorithms library
  <algorithm> Non-modifying sequence operations, Mutating sequence operations, Sorting and related operations
  <cstdlib> C library algorithms
Numerics library
  <complex> Complex numbers
  <valarray> Numeric arrays
  <numeric> Generalized numeric operations
  <cmath> <cstdlib> C library algorithms
Input/Output library
  <iosfwd> Forward declarations
  <iostream> Standard iostream objects
  <ios> iostreams base classes
  <iostream> <iostream> <iomanip> Formatting and manipulators
  <string> <cstring> <cwchar> File streams
Lexical layers of a C++ program
Namespaces
  Namespaces provide a way to avoid name conflicts
  Referencing a name x in another namespace called Marshall requires using Marshall::x
  Namespace definitions must be
  top-level
    in a namespace
  in a class
    locally in a function
Functions
  Variables and classes can be declared in function bodies
  They cannot be referenced outside the body of the function
  Functions can be declared
    top-level
    in a namespace
    As a member of a class (called a method)
Linkage - symbols have three kinds of "linkage"
  external - other translation units can see
    If two translation units have the same external symbol then they refer to the same thing
    The following default to external
      namespace scope function or reference
      namespace scope function
  internal - only same translation unit can see
    Even if another translation unit has the same symbol as external, it is a different symbol
    The following are internal
      namespace scope declared static
      namespace scope declared const and not extern
      namespace scope data member of anonymous union
  no linkage - can only be seen in the same scope
Kinds of things at runtime
Storage
  static storage
  heap storage
  stack frames
Storage duration
  static
  automatic
  dynamic (new/delete)
Kinds of data
  primitive values
  pointers
  class objects
    virtual function table
    function and method code bodies
Templates
  Templates allow defining a class or function that depends on a type that will be specified later
  When the class or function is actually used the type(s) must be specified
  How would one write a general function that swaps its arguments of any (identical) types?
  We will talk about templates later