C++ by examples

Jean-Louis Noullet - 2005 - rev. 3.0

1. Introduction

1.1 What is C++ ?

C++ is made of “C” and “++”

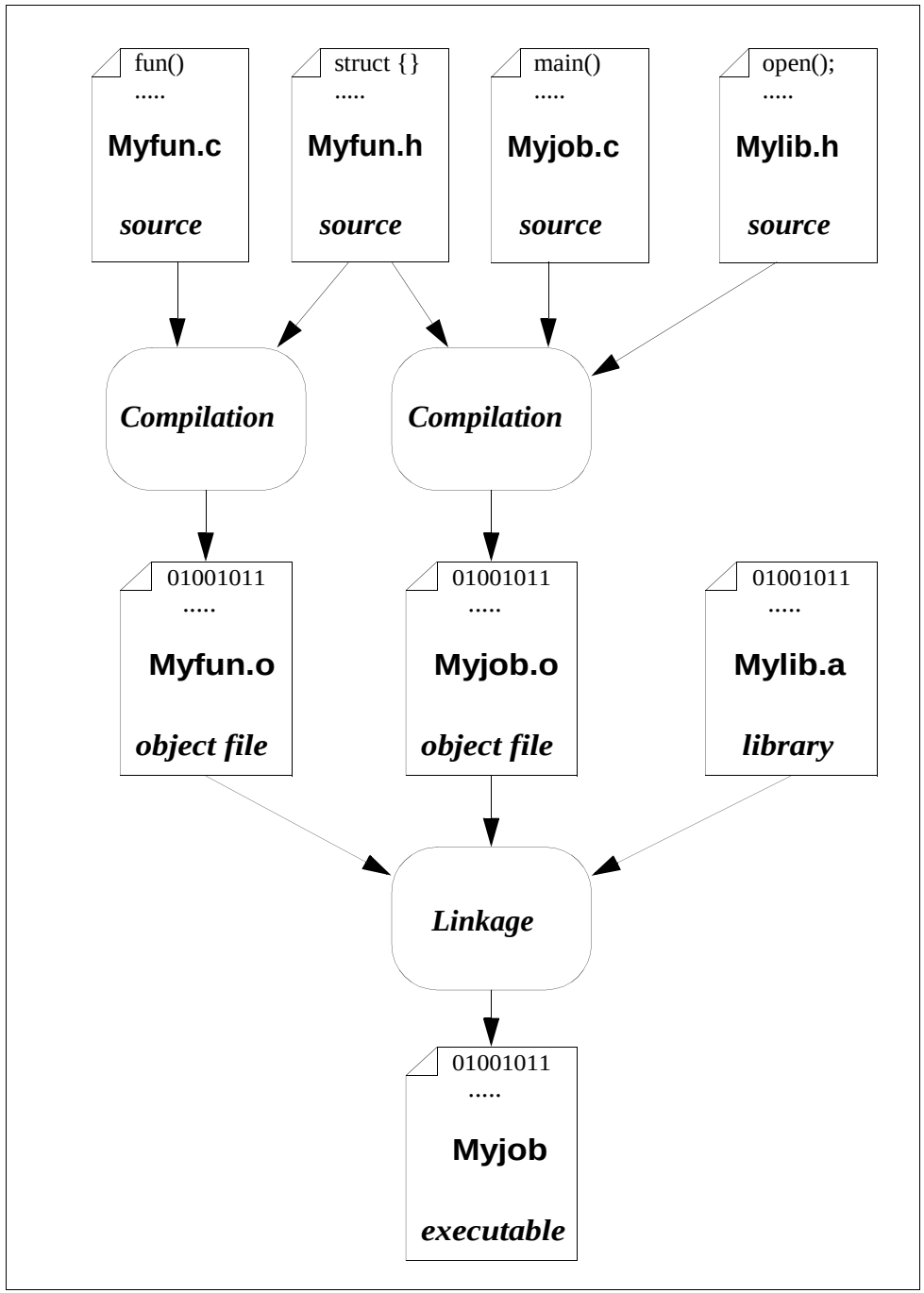
- C is a programming language which was created (by 1973) for facilitating the creation of a portable operating system : Unix (previous operating systems were written in assembly language, hardware dependant and expensive to develop).

- Its flexibility allowed to extend its application range from the small 8-bit microcontroller to the supercomputer, and it turned to be the “reference programming language”.

- C has a little number of built-in constructs, many things relying on standard libraries which come with the compiler.

- Some design flaws in the initial C spec (“K&R”) where corrected later by the “ANSI C” (1989), the only one in use now.

- C++ is an extension of C, created (by 1984) with the main goal of introducing object-oriented programming, while preserving the flexibility of C
- Another goal was to make software development “safer” than in C, by improving checkings.
- C++ is not a “pure object oriented” language, it allows mixing programming styles, for the best or the worse....
- C++ is hard to learn, because of the amount of knowledge involved (all the subtleties of C, plus all the features added by C++)
- C++ is a standard (ANSI then ISO), but not all implementation conform to the specification.
- C++ has a standard library, in addition to all the standard libraries of C



1.2 What about libraries ?

The making of a program involves 2 steps :

- the compilation itself, which produces “object code”, which is relocatable machine code
- the linking, which binds together pieces of object code to make an executable file

This allows “separate compilation”, the benefit of which is :

- saving compilation time
- possibility of mixing languages
- name space separation (to some degree)
- hiding implementation details, preserving IP

An ***object*** file contains the object code from a single source file.

A ***library*** contains the results of merging several object file and creating a kind of index allowing the linker to extract the functions needed for building the project.

A ***header*** file is a source code file containing only declarations required to make use of an already compiled object file or library. The header file is intended to be included in source code.

1.3 Compatibility

- C++ programs **can** call any library function created from C source code and assembly language source code (with convenient declarations i.e. extern “C” {})

- C source code **can** be merged into C++ code with some changes (ranging from no change to significant editing)

- C programs **cannot** call library functions created from C++ source code

1.4 Filename conventions

	Unix/Linux	Windows
source C	.c	.c
source C++	.cpp, .cxx	.cpp, .cxx
header C	.h	.h
header C++	(.h)	(.h)
object	.o	.obj
library	.a	.lib
executable		.exe

Note : the ‘.h’ extension was removed from standard C++ headers when namespaces were introduced.

1.5 Initial FAQ

- Does C++ allow faster development than C ?
  Yes, specially in large projects, because of
  - data abstraction facilitating code re-use
  - potentially safer memory management
  - built-in exception handling

- Does C++ make programs running faster (than C) ?
  - Not at all, except by the way it facilitates re-use of possibly efficient code
  - Unaware programmers may create very inefficient programs !

- Does C++ applications consume more memory (than C) ?
  - Generally yes, this is called “overhead”, because the compiler has to generate hidden code and data, and inheritance and templates hide complexity of objects

- How big is the effort of learning C++ ?
  - Huge, being fluent in C++ implies being fluent in C, plus object oriented programming, plus added libraries.

```
// file ex1.cpp
#include <iostream>
using namespace std;

// main function returning int
int main()
{
// h is a pointer to char, initialized with constant string
char * h = "hello ";
// i is an integer number
int i = 3;

cout << h << i << endl;
// this prints :
// hello 3

// the same results is obtained with :
( ( cout << h ) << i ) << endl;

return 0;
}
```

2. Preliminary

2.1 Stream output

The first feature to be introduced is the stream output. It is not fundamental, but we will need it for all the other examples : it is the tool for printing results to the console (and later to files).

The example involves **cout**, a predefined value of type **ostream** (from included header iostream) which point to standard output.

Obviously, the << operator concatenates pieces of text and sends them to cout, but what looks unusual (from the C programmer's point of view) is :

- it accepts incompatible types (**int** and pointer to **char**)
- it is evaluated from left to right, but **cout** is the leftmost side !

This involves 3 essential C++ features :

- function overloading
- operator redefinition
- parameter passing by reference

```
// file ex1.cpp
// function returning nothing, taking an int as argument
void display ( int n )
{ printf("%d", n ); }

// function returning nothing, taking a pointer to char
// as argument
void display ( char * c )
{ printf("%s", c ); }

// another one taking 2 arguments
void display ( char * c, int n )
{ printf("%s%d", c, n ); }

int main()
{
char * h = "hello ";
int i = 3;

display( h ); display ( i );
display ( "\n" );

display( h, i ); display ( "\n" );

return(0);
}
```

2.2 Function overloading

- declaration of function parameter types is mandatory (was optional in K&R C)
- the list of the types of the parameters is called the "function's **signature**"
- the signature together with the returned type form the "function's **prototype**"

Functions with :

- the same name
- different signatures

may exist simultaneously : this is function **overloading** (prohibited in C)

This is made possible by a modification of the object file format, called "name mangling" (signature is incorporated into the function's name).

A side effect is type checking at link time (improved safety).

In the example, function **printf** from the C standard library is involved, to obtain the same results as with previously with **cout** <<.

```
// file ex2.cpp
// function taking two pointers to int
void swap1( int * p1, int * p2 )
{ int tmp;
  tmp = *p1, *p1 = *p2; *p2 = tmp;
}

// function taking two references to int
void swap2( int & r1, int & r2 )
{ int tmp;
  tmp = r1, r1 = r2; r2 = tmp;
}

int main()
{
int i, j;
i = 13; j = 42;
cout << i << " " << j << endl;
// &i and &j are the addresses of i and j
swap1( &i, &j );
cout << i << " " << j << endl;
// looks like if we passed i and j by value...
swap2( i, j );
cout << i << " " << j << endl;

return 0; }
```

2.3 References

By default, function parameters are passed “by value”, which means that they are copied into the functions data space at the time of calling . These copies are lost when we return from the function.

If we want to avoid the copying, we may pass pointers to the variables instead of the variables themselves.
A pointer is a variable holding the address of another variable.

The indirection operator * allows to access to the data (reading or writing)

The address operator & returns the address of a variable, in order to assign it to a pointer.

C++ adds an alternative option : passing function parameters “by **reference**” : this also avoids copying, but we do not need to pass addresses explicitely, neither to use the indirection operator *.

Notice a new, different meaning of the & symbol...

```
// file ex2.cpp
// function returning a reference to int
int & tfind( int * table, int N,
             int val ) {
   static int trash;
   for ( int i = 0; i < N ; i++ )
       if ( table[i] == val )
          return table[i];
   return trash;
}

int main()
{
// array of integers, initialized with constants
int t[5] = { 1, 3, 5, 7, 9 };

// find and replace value 7 by 11
tfind( t, 5, 7 ) = 11;

int k = 0;
while ( k < 5 )
      cout << t[k++] << " ";
cout << endl;
return 0;
}
```

Reference vs pointer :

- a pointer is a distinct variable
- a pointer has a specific pointer type
- assigning to a pointer changes the pointer, not the data it points to
- some pointer arithmetics is legal (incrementing, adding an integer offset)

- reference is a “passing scheme” rather than a variable type
- a reference has the same type as the data it refers to
- assigning to a reference changes the data itself
- a reference is not a distinct variable, it behaves rather like an “alias” of an existing variable

The most important use of references is allowing a function’s returned value to be the left member of an assignment (“**lvalue**”), as in the example (would be impossible in C).

Do the references make pointers obsolete ?
No, pointers still have specific uses :

- the base of an array is a pointer
- pointers allow to store object addresses in arrays

```
// file ex3.cpp
#include <math.h>
#include <iostream>
using namespace std;

struct cplex {
double R;
double I;
};

inline double mdulus( cplex & c )
  {
  return( sqrt( ( c.R * c.R ) +
                ( c.I * c.I ) )
        );
  }

int main()
{
cplex A, B, C;
A.R = 1.2; A.I = 0.1;

cout << mdulus( A ) << endl;

return(0);
}
```

3. User defined type

3.1 Structure

Already used in C, structures are user-defined types. The syntax of declaring a structure type has changed in C++ (no more “typedef struct”).

In the example, we create a new type **cplex** to handle complex numbers (a complex type is available in the C++ library, but we create our own as an exercise).

R and I are **structure members** (aka fields).

A complete structure may be copied, but when passing it to function **mdulus**, we prefer to avoid an unecessary copy for efficiency reasons.

A short function may be created as “**inline**”, this allows the compiler to replace each fonction call by a flat copy of the function’s executable code. This affects only performance (may be faster).

```
// file ex3.cpp
// redefinition of the addition operator
cplex operator + ( cplex & a, cplex & b ) {
cplex res;
res.R = a.R + b.R; res.I = a.I + b.I;
return res;
}

// redefinition of the multiplication operator
cplex operator * ( cplex & a, cplex & b ) {
cplex res;
res.R = a.R * b.R - a.I * b.I;
res.I = a.R * b.I + a.I * b.R;
return res;
}

int main() {
cplex A, B, C;
A.R = 1.2; A.I = 0.1;
B.R = 2.3; B.I = 0.2;
C = A + B;

cout << C << endl;

return 0;
}
```

3.2 Operator redefinition

The C++ compiler converts operators into function calls, so there is an equivalent function for each operator.
Its name is composed of the keyword operator followed by the operator symbol.

Like any function, this equivalent function may be overloaded, by defining it with a different signature.

So it is possible to redefine operators for user-defined types.

Notice return values from the operator functions are passed by value (not by reference) in each case the return value is different from both arguments : here a copy is needed.

```
// file ex3.cpp
// redefinition of the increment operators
// increment-by
inline cplex & operator +=
       ( cplex & a, cplex & b ) {
   a.R += b.R; a.I += b.I;
   return a;
   }

// pre-increment
inline cplex & operator ++
       ( cplex & a ) {
   a.R += 1.0;
   return a;
   }

// post-increment (notice the dummy int)
cplex operator ++ ( cplex & a, int ) {
   cplex Old = a;
   a.R += 1.0;
   return Old;
   }
```

All operator functions have to return a value, even “++” which is sometimes used only for its side-effect.

If the function returns one of its arguments, the return may be passed by reference to avoid an unnecessary copy.

A trick was proposed in the C++ spec to allow a different function for pre-increment (like ++i) and post-increment (i++) : the trick is an additionnal dummy argument to make the signature different.

Both functions have the same side-effect but do not return the same value.

```
// file ex3.cpp
// redefinition of the sign operator (not subtract)
inline cplex operator - ( cplex & a ) {
   cplex res;
   res.R = -a.R; res.I = -a.I;
   return res;
   }

// redefinition of the output stream operator <<
ostream & operator <<
          ( ostream & out_stream, cplex & c )
   {
   out_stream << c.R << ":" << c.I;
   return out_stream;
   }

int main()
{
cplex A, B, C;
A.R = 1.2; A.I = 0.1;

C = -A;
cout << C << endl;
cout << A++ << endl;
return(0);
}
```

In the example, not all arithmetic operators are redefined, but they should...

The subtract operator '-' would be handled like the '+'.

Meanwhile, the '-' symbol is also used with a single argument to change the sign : another function is associated with this.

Now we see how the magic happens with the << operator :

- it is redefined for each type
- the output stream which is received is passed by reference (unchanged), so the function works only by side-effect.

The << operator works as stream operator only if its signature contains a stream argument, otherwise it may be used as another arithmetic operator (like bits shift for example).

```
// file ex4.cpp

class cplex {
public :
double R;
double I;

// inline member function
double modulus() {
  return( sqrt( ( R * R ) + ( I * I ) ) );
  }
...
};  // end of class declaration

int main()
{
cplex A, B, C;
A.R = 1.2; A.I = 0.1;

// member function called through object
cout << A.modulus() << endl;

return 0;
}
```

4. Class

4.1 Methods

Here begins object-oriented programming. Functions which are very specific to a user-defined type may be declared as members of the structure.

A type with **member functions** or **methods** is known as a **class** in O.O.P. (but in C++ a structure can do it as well)

A variable of such a type is an **object**.

Only member variables are stored in the data memory occupied by the object.

- member function names are in the class scope (like member variables) : no name conflict issue
- member functions are called through an object (unless “static”)
- member functions have direct access to member variables

```
class cplex {
public :
double R;
double I;
// inline member function
double modulus() {
  return( sqrt( ( R * R ) + ( I * I ) ) );
  }
// member functions prototypes
cplex operator + ( cplex & b );
cplex operator * ( cplex & b );
...
};  // end of class declaration

// member function bodies
cplex cplex::operator + ( cplex & b ) {
   cplex res;
   res.R = R + b.R; res.I = I + b.I;
   return res;
   }
// non-member function
ostream & operator <<
        ( ostream & out_stream, cplex & c ) {
   out_stream << c.R << ":" << c.I;
   return out_stream;
   }
```

- compared with non-member functions, member functions have typically less arguments, because of direct access to members.
- member functions may be defined in the class declaration, this implies inline.
- otherwise, only the function's prototype is in written the class declaration.

4.2 Scope resolution operator

- outside of the class declaration, the member names are not visible.
- The **scope resolution operator ::** allows to specify explicitely a name which is "out of reach". It is used for creating the member function bodies (unless inline)

4.3 Member operator functions

- in the case of an operator redefining function, if the **first** argument is of the current class type, in can be omitted and the function can be a member.
- if the first argument is another type, the function cannot be a member.

```
class cplex {
public :
double R;
double I;
...
cplex & operator += ( cplex & b ) {
  R += b.R; I += b.I;
  return *this;
  }

// pre-increment
cplex & operator ++ () {
  R += 1.0;
  return *this;
  }

// post-increment (notice the dummy int)
cplex operator ++ ( int ) {
  cplex Old = *this;
  R += 1.0;
  return Old;
  }

...
};    // end of class
```

4.4 “This” pointer

Member functions have direct, “implicit” access to the members of the current object.

Sometimes it is necessary to have an explicit access to this current objet.

The keyword “**this**” returns a pointer to the current object, the one from which the member function was called.

```
class cplex {
public :
double R;
double I;
// a variable for counting the cplex multiplications
static int cnt_mul;
...
static void reset_cnt() {
  cnt_mul = 0;
  }
};  // end of class declaration

// here storage is allocated in global space for our static member
int cplex::cnt_mul = 0;

int main()
{
cplex A = cplex( 2.0, 2.0 );
...
cout << cplex::cnt_mul << endl;
cout << A.cnt_mul << endl;
cplex::reset_cnt();
cout << A.cnt_mul << endl;
}
```

4.5 Static members and functions

A **static** data member is stored outside of any object, it is unique to the whole class, in O.O.P. it is a “class variable”.

It is necessary to define it in the global storage of some compilation unit (i.e. outside of any class or function).

It may be accessed by the scope resolution operator :: or through any object of the class, unless it is private.

Static member functions have only access to static members and functions.
They have no “this” pointer.

```
// file ex5.cpp

class cplex {
public :
double R;
double I;
// constructors
cplex () {
  R = I = 0.0;
  }
cplex ( double a ) {
  R = a; I = 0.0;
  }
cplex ( double a, double b ) {
  R = a; I = b;
  }
...
};

int main()
{
cplex A ( 1.2, 0.1 );
cplex B ( 2.3 );
cplex C;
...
}
```

5. Constructors

A class may have special member functions which are **constructors** and **destructors**.

When an object is created, the following happens :
- memory is allocated to the member variables
- a constructor is called (if any)

When an object is destroyed :
- the destructor is called (if any)
- member variables memory is freed

The constructor's name is always the name of the class.

5.1 Initializing the members

In the case of our cplex example, constructors may be usefull for initializing member variables from different sources.

Note : objects which are local variables in a function (like A, B, C in the example) are created when the function is called, and destroyed automatically when it is left (they belong to the "automatic" storage class).

```
int main()
{
cplex A ( 1.2, 0.1 );
cplex B ( 2.3 );
cplex C;
double D = 6.6;
int I = 1664;

// implicit conversion
C = D;
cout << C << endl;
// double implicit conversion
C = I;
cout << C << endl;

// explicit conversion, C++ style casting
C = cplex(D);
cout << C << endl;
// explicit conversion, C style casting
C = (cplex)I;
cout << C << endl;
...
}
```

5.2 Conversion from other type

A constructor taking exactly one argument may serve also for converting data from another type to the current type, it is a **conversion constructor**.

By default the conversion is implicit (no casting is required).

In the example, we see that implicit conversions may be cascaded :

- from int to double
- from double to cplex

This is powerfull but may hide programming errors, so we may prefer disable implicit conversions by declaring the constructor as **explicit** :

```
explicit cplex ( double a ) {
```

Conversions with explicit casting remain possible (and usefull).

```
// file ex5.cpp
class cplex {
public :
double R;
double I;
double modulus() {
  return( sqrt( ( R * R ) + ( I * I ) ) );
  }

// conversion operator from this type to other type
operator double() {
  return this->modulus();
  }
...
};  // end of class declaration


int main()
{
cplex A = cplex( 2.0, 2.0 );
cout << double(A) << endl;
cout << (double)A << endl;
}
```

5.3 Conversion to other type

The C++ style type casting operator may be redefined to handle conversion from the current class to some other type or class.

Here **double()** is not a function but an operator, since “double” is already known as a type.

Once the operator is defined, the C-style casting is accepted as well.

Danger : this operator cannot be declared as **“explicit”**, so one cannot prevent it to be involved in invisible implicit conversions.

A safer practice is to define **“accessors”** i.e. methods which can be used to explicitely convert from the current class to another type (here it could be getdouble(), but it is already named modulus()).

```
// file ex6.cpp
class cplex {
public :
double R;
double I;
...
// member functions not modifying the current object
double modulus() const {
  return(
        sqrt( ( R * R ) + ( I * I ) )
        );
  }
cplex operator + ( const cplex & b ) const;
cplex operator * ( const cplex & b ) const;

// member functions modifying the current object
cplex & operator += ( const cplex & b ) {
  R += b.R; I += b.I;
  return *this;
  }

};   // end class
```

6. Read-only data

6.1 Const objects

If a variable is declared with the **const** qualifier, the compiler checks that no attempt is done to change ist value, either :

- directly by assignment
- indirectly by function call

This feature allows to make an object read-only at some point of processing.

It makes sense only if each time a function never modifies one of its arguments, this fact is explicitely declared in the function's prototype (a "promise not to change data"). The **const** type qualifier is used for this purpose.

Of course this declaration is not necessary for arguments passed by value.

```
int main()
{
const cplex A ( 1.2, 0.1 );
cplex BB ( 1.3, 0.2 );
BB += A;

const cplex B = BB;
cplex C;

C = A + B;
cout << C << endl;
cout << A.modulus() << endl;

C = A * B;
cout << C << endl;

C = -A;
cout << C << endl;

C += B;
cout << C++ << endl;
return(0);
}
```

In the case of member functions, the current object is not in the arguments list, so the syntax to declare it read-only is putting the **const** keyword alone, after the arguments list.

In the example, the results of computations performed on object BB are stored safely in the read-only object B.

Any attempt to modify B would be detected at compilation time (even through a complex hierarchy of function calls).

```
// constant pointer to writable data
cplex * const P = &C;
// P = &A;  // refused
*P += B;
cout << *P << endl;

// pointer to constant data
const cplex * Q;
Q = &C;
// *Q += A;  // refused
cout << (*Q).modulus() << endl;

// a shortcut for accessing class member through
// a pointer (same results as previous line):
cout << Q->modulus() << endl;
```

6.2 Const pointers

Two different situations are possible :

- the pointer is read-only (but the data it points to may be modified)
- the data accessed through the pointer is read-only (but the pointer value (address) may change)

The syntax follows :

- type * **const** ptr // read-only pointer
- **const** type * ptr // read-only data

```
// file ex7.cpp
#include <string.h>
#include <iostream>
using namespace std;

// base class for Waves, has only real component
class Wave {
private :
  int size;     // usefull size
  int capa;     // allocated size
public :
  double *data;  // samples
  // default constructor
  Wave() {
     size = capa = 0;
     }
```

7. Larger objects

- Objects from a class may contain a variable amount of data, which calls for dynamic allocation.
- Dynamic allocation allows allocating memory blocks the size of which is known at run time only

In the proposed example, we begin a project involving sampled signals or “waves” on which we want to perform some signal processing tasks.

7.1 Private vs public members

- Each wave may contain contains a different number of samples
- The number of samples in a wave may be increased or decreased during processing
- So we decide to store separately the size of the wave and the currently allocated memory amount (greater or equal to the size)
- In order to prevent memory errors (like “segmentation faults” or “bus errors”) which could be caused by uncontrolled change on the size parameters, we make them **private**, which means accessible only to member functions (and non-member functions declared as “**friends**”).

```
// Wave class, continuation
  explicit Wave( int c ) {
     data = new double[c];
     capa = c; size = 0;
     }
  Wave( const double *d, int c ) {
     data = new double[c];
     capa = c; size = c;
     memcpy( data, d,
             c * sizeof(double) );
     }
...
}; // end class

int main() {
double wini[] = { 1.12, 2.23, 3.34 };

Wave s1;   // empty wave

Wave s2(3);   // allocated Wave
// Wave s2 = Wave( 3 );    // same result

Wave s3( wini, 3 );   // filled Wave
// Wave s3 = Wave( wini, 3 );
...
```

7.2 Constructing with “new”

In this example, when a Wave object is created, memory is automatically allocated for the members, which are two ints and one pointer to double (12 bytes on a 32-bit machine).
Further allocation must be done by the constructors.

The **new** operator allocates memory and returns a pointer, according to the specified type and optional array size.

It does more than *malloc* from C : when applied to a class type, it causes the constructor to be called.

In the example, two constructors are defined in addition to the default constructor.

The one with an int argument pre-allocates memory but does not put any data. It is declared “explicit” to prohibit implicit int-to-Wave conversion which could mask a programming mistake.

The last one initializes the Wave from a array of doubles, supplied with a size parameter.

```
// Wave class, continuation
...
// destructor
  ~Wave() {
     if ( capa )
        { delete [] data; }
     }
...
}; // end class


int main() {
double wini[] = { 1.12, 2.23, 3.34 };

wave * ps1;  // a pointer to a Wave

// dynamic allocating an object
ps1 = new Wave( wini, 3 );

...

// destroying the dynamically allocated object
delete ps1;
...
}
```

7.3 Destroying with "delete"

When an object is deleted (for example when the function where it was automatically created is left), its members which are pointers are lost, and the associated memory becomes unreachable.

It is necessary to free this memory before losing the pointers, otherwise "memory leakage" happens.

To do this, a destructor is defined, which calls the "**delete**" operator.

It is not necessary to pass the array size to "**delete**", but it must be called with '[ ]' if "**new**" was invoked for an array.

The destructor's name is always the class name with a '~' prefix.

When **delete** is called with a pointer to an object, the object's destructor is automatically called.

Note : **delete** should be always paired with **new.**

```
// Wave class, continuation
...
// copy-constructor
// (Note : capa is not copied, but set to the value of size
// for minimal allocation)
  Wave( const Wave & s ) {
     capa = s.size; size = s.size;
     data = new double[capa];
     memcpy( data, s.data,
           size * sizeof(double) );
     }
...
}; // end class


int main() {
Wave s3 = Wave( wini, 3 );
...
// creating s4 with the copy-constructor
Wave s4 = s3;
...
```

7.4 Copy constructor

The code generated by the compiler copies automatically the members of an object in two cases :

- construction of a new object initialized with an existing object,
- assignment using the '=' operator

In the case of objects containing pointers, the pointers are copied but the pointed data is not.
This results in two objects pointing to the same data, which may be harmfull :

- writing to one object's data modifies the other one
- deleting one after the other cause "delete" to fail

This can be solved by :

- defining a copy constructor
- redefining the assignment operator '='

The copy-constructor is just another constructor taking one argument of the current class type.

```
// Wave class, continuation
...
// assignment operator
  Wave & operator= ( const Wave & s ) {
     if   ( capa < s.size )
          {  // here resize is necessary
          if ( capa > 0 )
             delete [] data;
          data = new double[s.size];
          capa = s.size;
          }
     size = s.size;
     memcpy( data, s.data,
             size * sizeof(double) );
     return *this;
     }
...
}; // end class

int main() {
Wave s1;  // empty
...
// copying s3 to s1, expecting resize
s1 = s3;
...
```

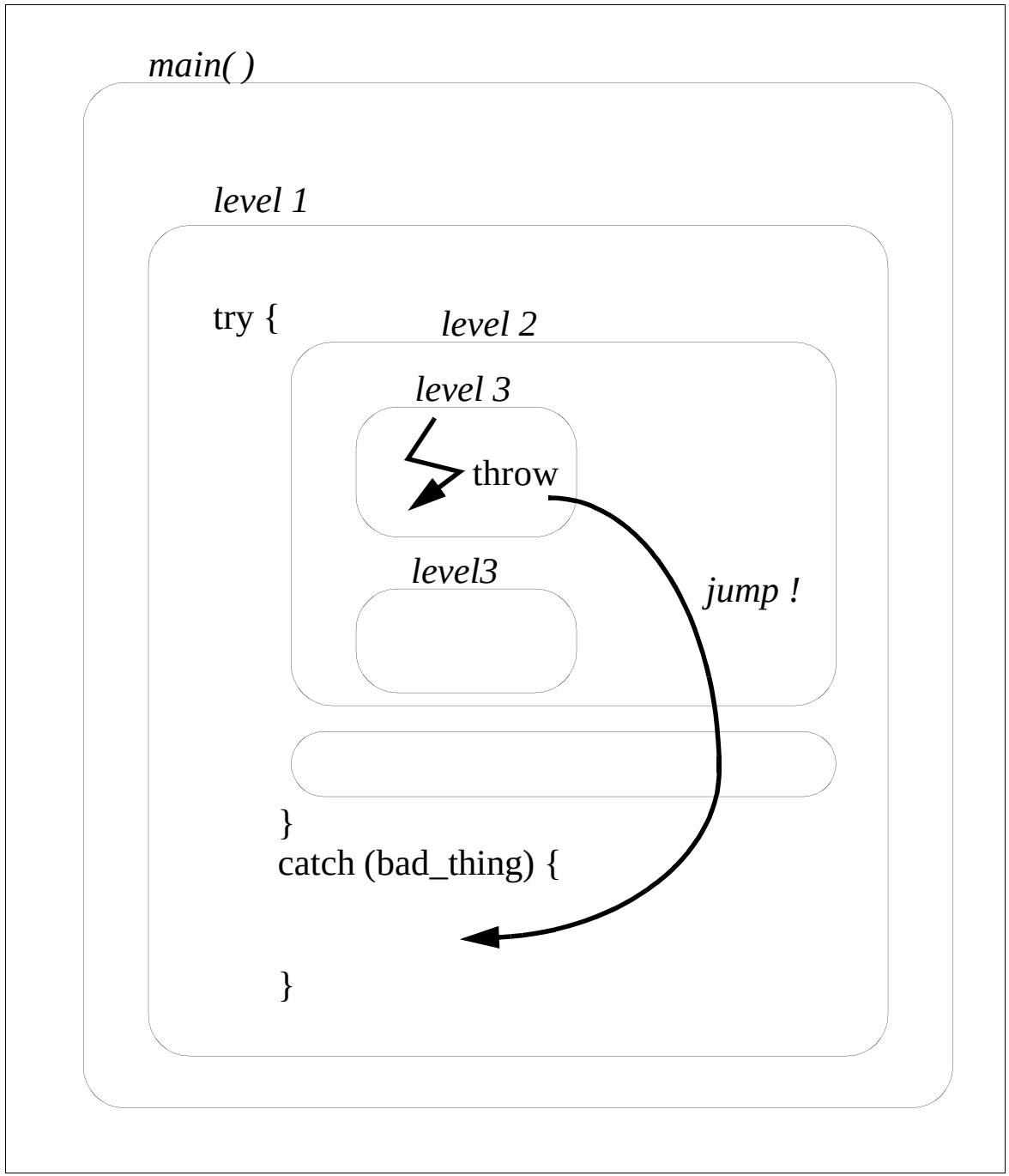
7.5 Assignment operator

In the case of objects involving dynamically allocated blocks, the '=' operator must be redefined in order to perform a complete copy of objects.

In the example, if the memory allocated in the destination object is not sufficient, it is resized before copying the samples.

Caution : even if the use of the '=' operator is not desired for the current class, redefining or disabling it is mandatory (as it is for the copy constructor), if the class has pointers to dynamically allocated memory. This is to avoid crashes that the default behavior of '=' would cause.

To disable **operator=**, just put its redefinition in a private region of the class.



8. Exceptions

Exceptions are unpredictable events which disrupt the normal flow of processing.

The simplest processing is just treating an exception as “fatal”, which consists of displaying an error message and exiting the program.

The most general solution is returning from the current function after putting an error code in the return value or in a global variable, deferring processing to the calling level. But this clutters a lot the large programs.

C++ introduces a built-in exception handling, which forces control to return from all the nested function levels until reaching a level where the exception is “caught”.

If the exception is not caught the program is aborted.

```
// file ex8.cpp

double * myfun2( int N ) {
  double *gross = new double[N];
  cout << "2\n"; return gross;
}

double * myfun1( int N ) {
  double *grass = myfun2( N );
  cout << "1\n"; return grass;
}

int main( int argc, char ** argv )
{
try {
  myfun1( 500000000 );
  } catch (bad_alloc &e) {
  cerr << "\n" << e.what() << "<--caught\n";
  } catch (...) { // catching anything
  cerr << "\n unknown exception caught\n";
  return(1);
  }
cout << "0\n";
return 0;
}
```

The exception may be thrown by a library function (standard libraries have predefined exceptions) or by a user routine.

An object is thrown, the type of which may be used to determine the origin of the problem.

In the example, we try to process the predefined "**bad_alloc**" exception thrown by the "**new**" operator in case of unsufficient memory.

The potentially failing code should be executed in a "**try**" block.
The **try** block is followed by one or more "**catch**" blocks. The **catch** block matching the type of the exception just thrown will be executed, and execution will continue after the last **catch** block.

The standard library exceptions have a method called "**what**" returning a short text.

```
#ifdef WIN32
#include <new.h>
int out_of_memory( unsigned int i) {
  std::bad_alloc b; throw b;
  return 0;
  }
#endif


int main( int argc, char ** argv )
{
#ifdef WIN32
// plug our own new exception handler
_set_new_handler(out_of_memory);
#endif

...
try {
...
    } catch (bad_alloc &e) {
...
    }
...
return 0;
}
```

In some implementations, the **new** operator does not throw any exception (but it should). Instead it returns a null pointer.

A solution could be testing the pointer and throwing the exception.

Another solution is using the **set_new_handler()** function to plug an exception processing routine in the **new** operator.

This solution is used in the example, in order to make VC++6 have a “normal” behavior, which is : throwing a **bad_alloc** exception when **new** fails.

(but with VC++6, we have to use _set_new_handler instead of set_new_handler ! :~()

The default behavior in presence of an uncaught exception may be customized by creating a terminate function plugged in by a call to **set_terminate()**. The terminate function must not return.

```
// file ex9.cpp
#include <iostream>
using namespace std;

// function template for swapping two objects
// of type T
template <typename T>
   void swap3( T & r1, T & r2 )
{ T tmp; tmp = r1, r1 = r2; r2 = tmp; }

int main()
{
double i, j;
i = 13.33; j = 42.22;
cout << i << " " << j << endl;

// instanciate template swap3 for double :
swap3 <double> ( i, j );
cout << i << " " << j << endl;
...
return 0;
}
```

9. Templates

Templates are function definitions or class definitions which are made generic by the use of one or more parameters.

Parameter values may be :
- type names or class names
- values of a simple type like int.

A parameter is like a “compilation-time variable” : parameters receive values at compile time.

The action of giving values to the parameter of a template is called “instanciation”.
The resulting function or class is an “instance” of the template.

The keyword **class** (old) or **typename** (modern) is used for defining a parameter holding the name of a type or a class.

```
// class template for a fixed size array
// takes a type and an int as parameters
// and parameters have default values
template <typename T = int, int SIZE = 100>
   class Vektor {
   public:
   T & operator[] (int i) {
       if ( ( i < 0 ) || ( i >= SIZE ) )
          i = 0;
       return buf[i];
       }
   int size() { return SIZE; }
   private:
   T buf[SIZE];
   };

int main()
{
// instanciate template Vektor with default size
Vektor <char> c;
c[0] = 'a';
c[99] = 'z';
c[200] = '!';
cout << c.size() << " ";
cout << c[-1] << c[0] << c[99] << endl;
...
```

In the example, a class for a “secure”, fixed size array of anything can be generated from the template “Vektor”.

In the class, the [ ] operator is redefined for type T, allowing the object to behave like an ordinary array.

The operator returns a reference, so that the return value can be used as a **lvalue**.

The index is checked, and any out-of-bounds access is redirected to element zero.

```
int main()
{
...
// instanciate another Vektor
Vektor <float, 3> F, G;
G[0] = 999;
F[0] = 0.11;
F[2] = 6.66;
F[200] = 3.14;
cout << F.size() << " ";
cout << F[-1] << " " << F[0] << " " << F[2]
<< endl;

// instanciate another swap3
swap3 < Vektor <float, 3> > ( F, G );

// swap3( F, G ) would just do the same by type deduction

cout << F[0] << endl;

return 0;
}
```

In the continuation of the example, two objects of another instance of Vektor are created.

Then they are permuted by an instance of the swap3 function, generated for type "`Vektor <float, 3>`".

Note : it is possible to instanciate functions template without specifying the parameters, if they can be derived from the arguments types ("type deduction").

```
// file ex10.cpp

class A {
public :
  A() : v(1) {
      cout << "constr A\n"; }
  A(int i) : v(i) {
      cout << "constr A\n"; }
  ~A() {
      cout << "destr A\n"; }
private :
  int v;
};

class B {
public :
  B() : am(), v2(0) {
      cout << "constr B\n"; }
  B(int i) : am(i), v2(i/2) {
      cout << "constr B\n"; }
  ~B() {
      cout << "destr B\n"; }
private :
  A am; // embedded object
  int v2;
};
```

10. Embedded objects

A class "B" may have objects of another class "A" as members, i.e. **embedded objects** or **member objects**.

The delicate point is constructing the embedded object:

- the "A" constructor is automatically called when allocating memory for members of "B"
- then control is passed to the constructor of "B", but it is too late to pass values to constructor of "A".

So a special construct called **initializer list** is introduced to initialize member objects explicitely before initializing the current object.
The syntax is generalized in such a way that simple members can also be initialized in this list.

In the example,

- class A constructors use the list for initializing the simple member **v**,
- class B constructors use the list for initializing the object member **am** and the simple member **v2**.

Destructors are always called automatically, in the reverse order.

base class	derived class
public	private
protected	private
private	unreachable
unreachable	

PRIVATE INHERITANCE

base class	derived class
public	protected
protected	protected
private	unreachable
unreachable	

PROTECTED INHERITANCE

base class	derived class
public	public
protected	protected
private	unreachable
unreachable	

PUBLIC INHERITANCE

11. Inheritance

The principle of object oriented programming is creating a hierarchy of classes, starting from the most generic and constructing more specialized classes by derivation.

A **derived class** inherits members and methods from its **base class** (or even several base classes).

Additional data members and methods are usually defined in the derived class. It is also frequent to create new methods which mask the method of the same name from the base class.

Accessibility management for inherited items involve a new access category which is "**protected**".

Protected members are like private for the outside world, but derived class methods have access to them.

Three inheritance schemes are available according to the tables. Public inheritance is frequently suitable, it gives to the derived class all the "capabilities" of the base class.

```
// file ex11.cpp

// base class
class A {
public :
  A() : v(1) { }
  A(int i) : v(i) { }
  void set( int i ) { v = i; }
  int get()  { return v; }
protected :
  int v;
};

// derived class
class B : public A {
public :
  B() : A(), v2(0) { }
  B(int i) : A(i), v2(i/2) { }
  void set( int i )
   { v = i; v2 = i*i; }
  int get2() { return v2; }
private :
  int v2;
};
```

Constructors are never inherited, but it is usual to invoke the base class constructor in the derived class constructor's intialization list, so they are re-used.

Destructors are not inherited, but base class destructors are called automatically when the derived class is destructed (after the base class destructor), so they are re-used too.

In the example, class **B** is publicly derived from class **A**. It has :

- two data members :
  - **v** , inherited from **A**
  - **v2**, new
- three methods :
  - **set**( ), new, masking the same inherited from **A**
  - **get**( ), inherited from **A**
  - **get2**( ), new
- two new constructors, but **A** constructors are invoked in the initialization lists

```
// file ex12.cpp
// base class
class A {
public :
  A() : v(1) { }
  A(int i) : v(i) { }
  virtual void set( int i )
   { v = i; }
  virtual int get()
   { return v; }
protected :
  int v;
};

// derived class
class B : public A {
public :
  B() : A(), v2(0) { }
  B(int i) : A(i), v2(i*2) { }
  virtual void set( int i )
   { v = i; v2 = i*i; }
  virtual int get()
   { return (v * v2); }
private :
  int v2;
};
```

12. Dynamic binding

When a class hierarchy is used, it is frequently convenient to put in an array (or some other container) pointers to various objects from classes derived from the same base. This allows to perform collective actions on object of mixed types having “something in common”.

Since these pointers have to be of the same type (the base class), objects “loose their identity”.

To allow objects to “keep their personality” even when accessed through generic pointers, we may use **dynamic binding** which is based on an underlying storage of type-related information in objects.

This is triggered by the presence of “**virtual”** functions in a base class. It results in the following :
“if a virtual function is redefined in a derived class, the redefined version will be called even if it is invoked from a pointer to the base class type”.

In other circumstances, virtual functions behave normally.

```
int main()
{
// table of pointers to base objects
A * too[3];

// creation of mixed-type objects
too[0] = new A(1);
too[1] = new A(2);
too[2] = new B(2);

cout << too[0]->get() << " "
     << too[1]->get() << " "
     << too[2]->get() << endl;

too[1]->set(3);
too[2]->set(3);

cout << too[0]->get() << " "
     << too[1]->get() << " "
     << too[2]->get() << endl;

return 0;
}
/* output is :
  1 2 8
  1 3 27 */
```

Here, we create an array "**too**" of pointers to **A** objects.

Then we initialize some elements with actual **A** objects, some others with **B** objects, **B** being a class derived from **A**.

When we call the method **get( )** for each element of this array "**too**", we obtain results which demonstrate that it is not the same **get( )** which was called in each case.

In fact the virtual function was chosen at run time, using indirection tables automatically created.

In large projects, objects are created only from derived classes, the base classes serving only as a code foundation or "**abstract classes**".

In this case the body of virtual functions in the base classes may be empty, which causes them to be called "**pure virtual functions**".

Dynamic binding also applies to destructors (in spite of their varying names). It is recommended to declare them as **virtual destructors**, since destruction may occur through a pointer (when calling delete).

```
// file ex13.cpp
#include <typeinfo>
A * too[3];
too[0] = new A(1);
too[1] = new A(2);
too[2] = new B(2);

cout << typeid(too).name() << endl;

cout << typeid(too[0]).name() << " "
     << typeid(too[1]).name() << " "
     << typeid(too[2]).name() << endl;

cout << typeid(*too[0]).name() << " "
     << typeid(*too[1]).name() << " "
     << typeid(*too[2]).name() << endl;
/* output from g++ :
  A3_P1A
  P1A P1A P1A
  1A 1A 1B

  output from VC++6 (with /GR) :
  class A * *
  class A * class A * class A *
  class A class A class B
*/
```

13. RTTI

"Run Time Type Identification" is a set of facilities derived from dynamic binding.

The **typeid** operator allows the program to query information about the actual type of objects.
It returns an objects, the methods of which allow :

- type comparisons
- text description
- etc...

In the example, which uses the same classes as in the previous one, **typeid** is applied to obtain a text description of the type of :

- the entire array of pointers to objects
- each element of the array
- each object pointed by these elements

In the third case, the actual type of objects is retreived.

```
// file ex14.cpp
#include <fstream>
#include <iostream>
using namespace std;

int main( int argc, char ** argv )
{

...
// creating ofstream object
ofstream report_file( argv[1] );
// checking (implicit conversion to boolean)
if   ( report_file )
     cout << argv[1]
          << " opened for writing\n";
else cout << "failed opening " << argv[1]
          << " for writing\n";

...
// writing to the file
int i = 1965;
report_file << "data = " << i << endl;

return 0;
// here file is automatically closed by ofstream class destructor
}
```

14. Stream I/O

Stream I/O applies to :

- console (keyboard and text screen)
- files
- pipes (from/to other processes)
- sockets (pipes through network)

The C++ standard library proposes a new concept of stream I/O as a replacement for the C library functions (fopen, gets, puts, fprintf, fscanf, etc...).
It features the stream operators << and >>.

14.1 File output

Creating an **ofstream** object with a text argument opens the file for output.

Opening failure does not cause an exception neither prevents the creation of the object.

ofstream is a derived class from **ostream**, this way the << redefinitions for **ostream** will work.

```
// file ex14.cpp continued
...
// creating ifstream object
ifstream config_file( argv[2] );
// checking (implicit conversion to boolean)
if   ( config_file )
     cout << argv[2]
          << " opened for reading\n";
else cout << "file " << argv[2]
          << " not found\n";

...
// “formatted” reading from the file
int j, k; double d;
config_file >> j >> k >> d;
```

14.2 File input

Operation of the **ifstream** object is symmetrical to the operation of **ofstream**.

The >> operator is available for standard types, and may be redefined for user classes.

It works for file input and for console input too, with the **cin** standard input stream.

In the example, the input file is supposed to contain 2 integer numbers and 1 floating number (all in decimal) separated by any amount of white space (space char, tabulations, newlines).

Uncorrectly formatted input causes no exception but loads unpredictable values in variables.

```
// file ex14.cpp continued
#include <iomanip>

cout << hex << 255 << endl;
cout << dec;  // back to default

cout << '*' << setw(4) << 12 << "*\n";

cout << '*'
     << hex << setw(4) << setfill('0')
     << 256+12 << "*\n";

cout << setprecision(2) << scientific
     << 1000.0/3.0 << endl;

cout << setprecision(2) << fixed
     << 1000.0/3.0 << endl;

/* expected printout :
ff
*  12*
*010c*
3.33e+02
333.33
*/
```

14.3 Formatting ouput

Customizing stream output format can be done by inserting manipulators in the flow. The manipulators cause a more or less persistent change in the state of the stream.

Some manipulators :

- **endl** : insert a newline char, then flush
- **flush** : flush buffers
- **hex** : print numbers in hexadecimal
- **dec** : print numbers in decimal
- **oct** : print numbers in octal
- **setw**(int n) : set the next field width
- **setfill**(char c) : set the fill character
- **left**, **right** : set justification
- **setprecision**(int n) : set number of significant digits after decimal point
- **fixed** : print reals in fixed point style
- **scientific** : print reals in scientific (exp) style

Inclusion of <iomanip> is required for the manipulators which take arguments

Custom manipulators may be created.

```
// file ex14.cpp continued

// write 1 char
report_file.put( '!' );

// write the contents of a buffer
char * t = "hello, people";
report_file.write( t, 5 ); // just hello
report_file.put( char(10) ); // newline

// read one line (leaving the newline char)
char lbuf[25];
config_file.get( lbuf, 25, '\n' );
cout << lbuf << '*' << endl;
// read one char
char c;
config_file.get( c ); // passed as reference!
cout << int(c) << '*' << endl;

// read all remaining lines
while ( config_file.getline( lbuf, 25 ) )
   {
   cout << '<' << lbuf << '>' << endl;
   };
```

14.4 Unformatted I/O

Methods exist for lower level reading or writing of :

- single characters or bytes
- lines of text
- records separated by a given delimiter

get and **getline** accept a custom delimiter (default is \n)

get leaves the delimiter in the stream, **getline** removes and discards it.

To avoid memory faults, **get** and **getline** take the buffer size as second argument (they read size-1 characters due to the trailing null which they append).

Trouble may be caused by DOS/Windows text files with pairs of CR/LF characters at the end of each line : **getline** keeps the CR !

read and **write** allow passing raw binary data (beware of portability issues !).

```
// file ex14.cpp continued

// rewind the input file
config_file.seekg( 0 );
// read 3 first bytes
char bbuf[3];
config_file.read( bbuf, 3 );
cout << bbuf[0] << bbuf[1] << bbuf[2]
     << endl;

// go close to the end
config_file.seekg( -3, ios_base::end );
// read 3 last bytes
config_file.read( bbuf, 3 );
cout << bbuf[0] << bbuf[1] << bbuf[2]
     << endl;
```

14.5 File seeking

The disk file I/O system functions (which are wrapped in the C and C++ library functions) maintain a current position pointer to each opened file.

Seeking means modifying explicitely this pointer, which makes sense only for binary file or text files with fixed length records.

Seeking is not allowed for console I/O, pipes, and other “non-seekable” media, and has uncertain compatibility with higher level functions like getline.

The second argument to **seekg** may be one of the three options :

- ios_base::beg (relative to begin of file)
- ios_base::cur (relative to position)
- ios_base::end (relative to the end)

```
// file ex15.cpp
#include <string>
using namespace std;

// initializing from ASCIIZ
string s1 = "hello world";
cout << s1 << "  " << s1.size() << endl;

// assigning from ASCIIZ
s1 = "good morning";
cout << s1 << "  " << s1.size() << endl;

// concatenation
string s2(" folks");
s1 = s1 + s2;
s1 += " !";
cout << s1 << "  " << s1.size() << endl;

// access to 1 char
char c = s1[3];
cout << "(" << c << ")" << endl;

// conversion to ASCIIZ (no implicit)
const char * oldstring;
oldstring = s1.c_str();
printf("{%s}\n", oldstring );
```

15. Strings

In C, text strings are represented as arrays for char terminated by a null char (zero-terminated-string, also known as ASCIIZ). This representation is also used in system functions in Unix and Windows.

Its use requires much precautions to avoid memory faults.

The C++ standard library proposes a new approach, the **string** class, where memory allocation is safely automated.

Appending strings is easy thanks to the redefinition of operators + and +=.

Implicit conversion from ASCIIZ works, for conversion to ASCIIZ, method **c_str**() must be used explicitely.

Many operators and methods are available to work with strings, and to mix strings and ASCIIZ text.

```
// file ex15.cpp continued
#include <string>
#include <sstream>
using namespace std;

// formatted output to a string
ostringstream os;
os << hex << 256+13 << " and more";
s1 = os.str();
cout << s1 << "  " << s1.size() << endl;

// number extraction from text
istringstream is("0.00000000314");
double x;
is >> x;
cout << x << endl;
```

stringstreams are i/o streams which store data in memory instead of communicating with a peripheral.

This allows to take benefit from the stream system, for example to :

- perform formatted output to a string (like good old sprintf).
- extract numerical values from text (like good old sscanf, atoi, atof)

16. STL overview

The **S**tandard **T**emplates **L**ibrary was initially a separated add-on to the C++ development tools.
Now it is included in the ISO specification and shipped within the C++ standard library.

The STL is based on generic **concepts**, which are mainly :

- **containers** : objects holding a variable number of user-defined objects
- **iterators** : generalized pointers allowing traversing containers
- **algorithms** : non-member functions performing actions on containers or slices (ranges) taken in containers

In addition the STL offers a number of **utilities**, which are used inside the STL templates but may also be used by the developer.

The main containers are :

- sequence containers : **vector**, **list**, **deque**
- container adaptors : **queue**, **stack**, **priority_queue**
- associative containers : **set**, **map**, **multiset**, **multimap**

They offer services which may be found as built-in constructs in other languages (Lisp, Perl, etc...)

Sequence containers are ordered collections, which have a beginning (or **front**) and an end (or **back**).
Container adaptors are created on top of the sequence containers to give more specialized access capabilities.
Associative containers are based on **keys** of arbitrary type allowing a pure random access (anyway they have also an internal ordered structure)

Note : “concept” has no formal implementation in C++. It is rather a hierarchy of design solutions, like “patterns”.

```
// file ex16.cpp
#include <iostream>
#include <vector>
using namespace std;

int main () {
// creating a container
vector<int> v;
// putting elements in the container
// (dynamic memory allocated automatically)
v.push_back( 11 );
v.push_back( 13 );

// reading with an iterator
vector<int>::iterator p;
  for ( p = v.begin();
        p != v.end();
        p++ )
      cout << *p << " ";
  cout << endl;

return 0;
}
```

16.1 Iterators

An **iterator** is a generalized pointer, an object which at least can be :

- dereferenced using the indirection operator ***** (and also -> if relevant)
- incremented with ++ to reach the next element
- compared with == and **!=**

It is more abstract than a pointer (++ does not mean necessarily “increment the address”)

The iterator types are nested types created within the containers classes.

Container classes offers **begin()** and **end()** members returning initial and final values of a traversing iterator.

Note : **end()** points “just after” the last element, i.e. on a ***virtual*** element which would follow the last real one.

Some benefits of iterators :

- iterators work similarly with all containers
- algorithm functions acccept iterators as arguments and may be automatically parametrized by them.

Warning : some container operations may **invalidate** a previously stored iterator.

```
// file ex16.cpp continued

struct spy {
spy() { cout << "constr.\n"; }
spy( const spy & s )
  { cout << "copy constr.\n"; }
~spy() { cout << "destr.\n"; }
spy & operator = ( const spy & s )
  { cout << "copy\n";
    return( *this );
  }
};

// a function to show a container as auto storage object
void myfun() {
list<spy> v;
v.push_back( spy() );
v.push_back( spy() );
cout << "myfun ready for return\n";
}

int main () {
...
cout << "call to myfun\n";
myfun();
cout << "returned from myfun\n";
```

16.2 Memory allocation service

Containers provide a consistent dynamic memory allocation service, which includes :

- allocating memory and calling constructors when new elements are insterted in or appended to the container
- destructing and freeing memory when the container is destroyed or shrunk.

Inserting or appending an object to a container involves always making a copy of it (like method **push_back()** does).
This may affect performance in the case of big objects.

A frequent solution consists of putting only pointers in the container, but in this case allocating and freeing data memory is no longer automatic (a pointer has no constructors neither destructors).

More sophisticated solution are sometimes proposed as “smart pointers”.

17. STL Sequence Containers

The sequence containers have in common many methods like :

- size_t **size()** : returns the number of elements
- size_t **max_size()** : returns the max size (a architecture dependant constant, does not reflect the available memory)
- bool **empty()** : tells wether the container is empty
- void **clear()** : make the container empty
- void **resize(**size_t new_size, T init_value**)** : resizes and fills at the same time
- T & **front()** : returns a ref to the first element
- T & **back()** : returns a ref to the last element
- void **push_back(** T & value**)** : appends a new value
- void **pop_back()** : removes the last element (but does not return it, use **back()** before !)

They also support common iterator-oriented methods :

- **begin()** : returns an iterator on the first element
- **end()** : returns an iterator pointing “just after” the last element
- **insert(** *iterator*, T & value **)** : inserts one elements of given value at position pointed by iterator
- **insert(** *iterator*, size_t number, T & value **)** : inserts several elements of given value
- **erase(** *iterator* **)** : erase an element (eventually shifting others)

In applications based on this common framework, container type may be easily changed from one to another (among vector, list, deque) for performance tuning.
Note : in the above, **size_t** is a standard type for sizes (usually equivalent to **int**), **T** is the type of the current container’s elements, *iterator* is the iterator type of the current container, like `vector<int>::iterator`

```
// file ex171.cpp
#include <vector>
using namespace std;

int main () {
vector<int> v;
v.push_back( 11 ); v.push_back( 13 );

// reading with subscript (writing also possible)
unsigned int i;
for ( i = 0; i < v.size(); i++ )
    cout << v[i] << " ";
cout << endl;

// reading with checked subscript
// (writing also possible)
try {
    for ( i = 0; i <= v.size(); i++ )
        cout << v.at(i) << " ";
    cout << endl;
    } catch ( exception & e )
         {
         cout << e.what() << endl;
         }
return 0;
}
```

17.1 The vector

The vector is much like an array in C, it is optimal for indexed access (aka subscript access).
For that purposes, it comes with a redefined **[]** operator.

A method called **at()** provides indexed access too, but with an additionnal bounds check which is not done by operator **[]**. (In case of an out-of-bounds access, **at()** throws a standard exception)

Warning : writing to an out-of-bounds index does not automatically increases the size of the vector (unlike with Perl arrays, unlike with C++ map).

The size of the array is modified by push_back(), pop_back(), resize(), insert(), erase(), etc..
One cannot increase the size without writing some value (no undefined entries).

Inserting or erasing in the middle or at the beginning of the vector is relatively slow, it is said “done in linear time” (time proportional to the vector size).

```
// file ex171.cpp continued

int main() {
...

unsigned int old_capacity = 0;

// optional pre-allocation
v.reserve(5);

// we fill the vector while watching capacity
for ( i = 0; i < 520; i++ )
    {
    if ( v.capacity() != old_capacity )
       {
       cout << "vector capacity = "
            << v.capacity() << endl;
       old_capacity = v.capacity();
       }
    v.push_back( i );
    }
```

Since the vector is kept in memory as a single linear array, increasing its memory capacity implies allocating a new array, copying the elements one by one and freeing the old storage. This is automatic, but for performance tuning the following methods are available :

- size_t **capacity()** : returns the currently allocated capacity (greater than or equal to size)
- **reserve(** size_t new_capacity**)** : allocates or reallocates to a new capacity, without changing size

When left automatic, allocation is usually done according to some “exponential” progression.

This progression can be shortened by a initial call to **reserve()**.

Danger : iterators may be **invalidated** by any change in the array’s size or capacity.

```
// file ex172.cpp
#include <iostream>
#include <list>
using namespace std;

// reading with an iterator
// (note the keyword typename used to remove ambiguity)
template < typename T >
   void dumplist( list<T> & lis )
   {
   typename std::list<T>::iterator p;
   for ( p = lis.begin();
         p != lis.end(); p++ )
         cout << *p << " ";
   cout << endl;
   }

int main () {
// creating a container
typedef list<int> mylist;
mylist li;
// putting elements in the container
li.push_back( 11 );  li.push_back( 13 );
li.push_front( 25 ); li.push_front( 50 );

dumplist<int> ( li );
```

17.2 The list

The **list** container is a doubly linked list.
(a singly linked list **slist** is also available)

The benefits of linked lists are :

- increasing the allocated memory does not imply moving the existing elements
- inserting or removing elements does not imply moving the others elements
- doing all this does not invalidate iterators (except may be one)
- insertion/removal at any place is done in constant (short) time

But the price to pay is :

- subscript access is not provided (would be quite inefficient)

```
// file ex172.cpp continued

// inserting after the i th element :
unsigned int i;
mylist::iterator pos;
pos = li.begin(); i = 3;
while ( i )
    { if ( ++pos == li.end() ) break;
      i--; }
li.insert( pos, 999 );

dumplist ( li );

// testing the containers's copy constructor
mylist la = li;
// testing the container's operator =
mylist lo;
lo = la;
dumplist ( lo );
// sort the copy
lo.sort();
// splice it into original list at pos
li.splice( pos, lo, lo.begin(), lo.end());
dumplist ( li );
// where are lo's elements now ?
cout << lo.size() << endl;
```

Some additional methods are :

- void **push_front(** T & value**)**
- void **pop_front()**
- **splice()**
- **sort()** : requires the existence of a '<' operator for the elements type.

The example shows the slow (but only) solution for reaching the "i th element".

Then it is verifyed that the list (like the other STL containers) has a copy constructor and an assignment operator.

The most general form of **splice()** is demonstrated, inserting at a given position (indicated by an iterator) a slice of another list (indicated by a pair of iterators).
Note that **splice()** only moves data, no copy is done.

One can also notice that using **insert()** at *pos* left this iterator pointing at the same element than before (but not at the same distance from front).

```
// file ex173.cpp
#include <iostream>
#include <deque>
using namespace std;
// reading with a subscript
template < typename T >
  void dumpcont( T & cont ) {
  for ( unsigned i = 0; i < cont.size(); i++ )
     cout << cont.at(i) << " ";
  cout << endl;
  }

int main () {
// creating a container
typedef deque<int> mydek;
mydek dk;
// putting elements in the container
dk.push_back( 11 );  dk.push_back( 13 );
dk.push_front( 25 ); dk.push_front( 50 );
dumpcont ( dk );

// inserting after the i th element :
mydek::iterator pos = dk.begin() + 3;
dk.insert( pos, 999 );

dumpcont ( dk );
```

17.3 The deque

The STL **deque** (from **d**ouble **e**ntry **que**ue) is an attempt to have the benefits of the vector and the list in the same container.

To summarize, the **deque** behaves like a **vector** which :

- supports **push_front()** and **pop_front()**
- performs insertions much faster
- has no **capacity()** nor **reserve()**
- grows without having to copy its whole contents from time to time

This is obtained at the expense of some general time and memory overhead.

If the total storage size is predictable, **vector** with **reserve()** is better.

If the storage need is unpredictable and/or many insertions are to be done, **deque** is better.

Note : as seen in the example, **deque** like **vector** supports iterator arithmetics (“random access iterator”).

18. STL container adaptors

The adaptors are created on top of the sequence containers, to provide some commonly used container patterns.

18.1 The stack

The stack adaptor implements the **LIFO** storage (Last-In, First-Out). Its specific methods are :

- void **push(**T &**)** : push a new element on top
- T & **top()** : retrieve the top element
- void **pop()** : remove (discard) the top element

Note : top() and pop() are separated functions for efficiency reasons (top returns by reference)

18.2 The queue

The queue adaptor implements the **FIFO** storage (First-In, First-Out). Its specific methods are :

- void **push(**T &**)** : push a new element at the back
- T & **front()** : retrieve the front (oldest) element
- void **pop()** : remove (discard) the front element

18.3 The priority queue

The priority queue adaptor implements a so-called heap storage, where elements are maintained sorted each time a new one is inserted. The first-out element is the greatest.

- void **push(**T &**)** : push a new element (automatically inserted where necessary to maintain order)
- T & **top()** : retrieve the greatest element
- void **pop()** : remove (discard) the greatest element

```
// file ex191.cpp
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
// a map indexed by strings, containing numbers
map < string, double > price;
price[string("potato")] = 0.25;
price[string("ham")] = 5.25;
price[string("wine")] = 4.25;
price[string("potato")] = 0.55;

cout << price.size() << endl;

// strange behavior : this creates a key-value pair
double d = price[string("beer")];
cout << price.size() << endl;

return 0;
}
```

19. STL associative containers

An associative container is a variable size container which supports efficient retrieval of elements values based on keys (like a dictionnary).
The container is parametrized by two types, the value_type and the key_type.
The key_type must have a comparison operator like < or a comparison function defining a strict weak ordering.

The key search is much faster than would be a linear search (logarithmic time).

19.1 The map

The map is a collection of key-value pairs.
Each key is unique and non-mutable.

The map supports a subscript access with operator [] accepting a key as subscript.
Writing with an unexisting key creates a new key-value pair, while writing with an existing key just overwrites the value.

```
// file ex191.cpp continued

// function accepting a range of iterator
template < class itero >
  void dumpmap ( itero begin, itero end ) {
  while ( begin != end ) {
     cout << begin->first << "==>"
          << begin->second << endl;
     begin++;
     }
  }

int main() {
...
// check key existence before reading value
if ( price.count(string("beans")) )
   {
   d = price[string("beans")];
   cout << string("beans") << "-->"
        << d << endl;
   }
// delete a key
if ( price.count(string("beer")) )
   price.erase( string("beer") );

dumpmap( price.begin(), price.end() );
```

The map supports iterators. Dereferencing the iterator yields a pair, which is a structure containing :

- a member called **first** : the key
- a member called **second** : the value

The existence of a key may be checked by :

- the method **count()** (returns 0 or 1),
- the method **find()** (returns an iterator on the pair found, or **end()** if not found)

Note : reading at an unexisting key never fails, but creates the key !
Always test the existence of the key before trying to read the value.

Some STL implementations propose also a **hash_map**, which differs from map only by the internal sort algorithm (hash table).

```
// file ex192.cpp
#include <iostream>
#include <set>
using namespace std;
// function accepting a range of iterator
template < class itero >
  void dumpset ( itero begin, itero end ) {
  while ( begin != end ) {
     cout << *(begin++) << " ";
     } cout << endl;
  }

int main() {
int table[] = { 13, 37, 19, 37, 4 };
// initializing a container from an iterator range
// C pointers are legal iterators !
set<int> ensemble( table, table+5 );

// adding an element
ensemble.insert( 122 );
dumpset( ensemble.begin(), ensemble.end() );
// checking existence
if ( ensemble.count(19) )
   cout << "19 exists\n";

return 0; }
```

19.2 The set

The **set** is like a **map** without values : only keys are stored.

It is used each time one needs to be able to check quickly wether a key is or is not in a collection.

The **insert()** method is used to add elements to the set.
Keys are unique in the set.
Inserting an existing key does nothing.

The existence of a key may be checked by :
- the method **count()** (returns 0 or 1),
- the method **find()** (returns an iterator)

19.3 multimap and multiset

multimap and multiset accept more than one element with a given key.
- The subscript operator **[]** is not available
- **count()** may return more than one
- methods **lower_bound()** and **upper_bound()** return an iterator range as result of search for a given key.

```
// file ex20.cpp
#include <iostream>
#include <algorithm>
using namespace std;

// custom comparison function
// dictionnary order wanted
bool mycomp( char c1, char c2 ) {
char lc1 = tolower(c1);
char lc2 = tolower(c2);
if ( lc1 == lc2 ) return( c1 < c2 );
return( lc1 < lc2 );
}

int main() {
char table[] = "aZeRtuUioTvWcxe";
const int N = sizeof( table ) - 1;
// C pointers are legal iterators !
sort( table, table+N, mycomp );

cout << table << endl;
// expected : aceeioRTtUuvWxZ

return 0;
}
```

20. STL algorithms

The algorithms are non-member functions which generally operate on ranges (couples of iterators) rather than on containers.

Many algorithms may call a user-defined callback function, which may be passed as a function pointer or as a function object.

A function object is an object for which the ( ) operator is redefined.

The example shows the use of the **sort** algorithm with a custom comparison function overriding the types's < operator.

Thanks to the fact that C pointers are legal iterators, the algorithm works on a simple array of char.