

# 00. Program (1)

# program

a **text file** is the **unit of storage** and the **unit of compilation**.

Non-trivial C++ programs are always split in **several files** for **good programing style** (modularity) and the use of **libraries**.

There are **2 ways** to **implement programs** :

- you can do everything yourselves **manually**: edition of files, compilation, link, *etc.* It is **hard** but you **master** what you are doing,
- or you can use an **integrated development environments** (IDE), like **eclipse**. An IDE will help you for compiling and linking a large number of files. The programmers can spend more time on code.

**select** your **preferred** method.

## program : **start** and **termination**

a **program** contains a **unique global function** called "**main**" which is the **starting point** of the program.

The "**main**" function must return a value of type **int** otherwise its type is **implementation-defined**

every **compiler** allows the following **definitions** of **main**.

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

When the "**main**" function terminate, the program terminate too.

# Write your first c++ program using **eclipse**(1)

- run **eclipse**
- in the **Workspace launcher**, give the name of the **workspace** where **eclipse** will store your programs
- in the **welcome** window, **click** on **go to the Workbench**
- in the **File** menu, **choose** the **New** menu and **click** on **Project...**
- In the **Select a wizard** window, **choose** the **C/C++ sub-menu** and **click** on **c++ Project**

# Write your first c++ program using **eclipse**(2)

- click **next**
- give your **project** a **name**
- in the **Project Type** menu, **click** on the **Executable** sub-menu and choose **Hello World c++ Project**
- click **Finish**
- **accept** the **C/C++ perspective**
- in the **left window**, **open** the **src** sub-menu and **double click** the **name** of your **program.cpp**

```
#include <iostream>
using namespace std;

int main() {
    cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
    return 0;
}
```

# Description of this program(1)

**comments** begin with `//` and terminate at the end of the line

**iostream** is an **already-implemented code** to deal with **input** and **output streams**

**#include** is a pre-processor directive (`#`) to insert code in your programs

**iostream** being included between `< >` mean that it is implemented inside the **standard C++ library**

C++ **knows** where to find the library containing this code

## Description of this program(2)

every function implemented inside the standard library must be prefixed by **std::** when used

**cout** and **endl** are **functions** implemented in the **iostream** *library*

you don't need to **prefix** cout and endl by std:: because you are **using** the **std namespace**

```
#include <iostream>
int main() {
    std::cout << "!!!Hello World!!!" << std::endl;
    return 0;
}
```

this **program** returns a **0 value** to its **caller**



# Calling your program

**click** on your **Project** name

**inside** the **bottom** window the text **!!!Hello World!!!** appeared

if something seems wrong :

- **left click** on the **name** of your **project** and choose **Refresh**
- open the menubar **submenu** and choose **Clean**

# command-line arguments

A program can have **arguments**, this arguments are passed to the program with **argc** and **argv**.

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

**argc** is the **number** of arguments passed to the program. When **argc** is nonzero:

- **argv** is an **array** of **strings** from `argv[0]` to `argv[argc-1]`
- the first argument `argv[0]` is the name of the program,

# main function properties

- the value of **argc** is non-negative
- the value of **argv[argc]** is 0
- the function **main** cannot be called within a program
- a program that declare **main inline** or **static** is ill-formed
- functions, member functions, classes, enumerations, *etc.* can be used in **main**

# Print the number of the main command-line arguments

```
#include <iostream>
using namespace std;
int main (int argc, char* argv[]) {
    cout << "The number of arguments is " << argc << endl;
    return 0;
}
```

# 01. **Variables** and **Types**

# Variables

Every variable has an **identifier** or name and a **type**.

The **identifier** allow using the variable.

The **type** determines :

- what **operations** can be **applied** to the variable
- the **behavior** of this operations

# Fundamental Types

C++ define the following **fundamental types** :

- `int`, integer value
- `char`, integer value and ascii
- `bool`, boolean, can be `true` or `false`
- `float`, single-precision floating point
- `double`, double-precision floating point
- `void`, type for not typed value
- *(and pointer)*

The size and range of each fundamental type is **implementation-defined**. For simplification reason I will use gcc size choice in 32 bits, a common choice.

# Range and Size of Fundamental Types

Common size and range for 32 bits architecture.

- `int` are in  $[-2^{31}, 2^{31} - 1]$ , `sizeof(int) == 4`
- `char` are in  $[-128, 127]$  ( $[-2^7, 2^7 - 1]$ ), `sizeof(char) == 1`
- `bool` can `true` or `false`
- `float` are  $[-3.40282e + 38, 3.40282e + 38]$ , `sizeof(float) == 4`
- `double` are  $[-1.79769e + 308, 1.79769e + 308]$ , `sizeof(double) == 8`
- `void` nothing can be affected to void.



# Sign modifier of Fundamental Types

range of fundamental type are **implementation-defined** but we can change it a little bit. For `int` and `char` following sign modifier can be applied:

- `signed`, enforce signed value.
- `unsigned`, enforce unsigned value.

Sign modifier enforce the sign of fundamental type and change the range of value as following:

- `signed int` are in  $[-2^{31}, 2^{31} - 1]$
- `unsigned int` are in  $[0, 2^{32} - 1]$
- `signed char` are in  $[-128, 127]$
- `unsigned char` are in  $[0, 255]$

# Size of Fundamental Types

Size of `int` type is **implementation-defined** but can be modified with following size modifier:

- `short`, integer have a reduced size and range
- `long`, integer have a larger size and range

This modifier are **implementation-defined** but follow this rules:

$$\begin{aligned} \text{sizeof}(\text{short int}) &\leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long int}) \\ \text{short int} &\subseteq \text{int} \subseteq \text{long int} \end{aligned}$$

Size modifier modified `int` and change the range of value as following:

- `int` are in  $[-2^{31}, 2^{31} - 1]$
- `short int` are in  $[-2^{15}, 2^{15} - 1]$
- `long int` are in  $[-2^{31}, 2^{31} - 1]$

The `long` modifier can be applied to **double**.

# Modifier combination and sugar form

Sign and size modifier may be combined. For example:

- `unsigned short int`
- `signed long int`

This declaration is quite long and C++ allow a shorter form so you can use:

- `unsigned short` for `unsigned short int`
- or `signed long` for `signed long int`
- and so on.

# Operations over fundamental type

The fundamental type have following operation available:

+	-	*	/	%	^	&		~	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
=	<<	>>	>>=	<<=	==	!=	<=	>=	&&
	++	--							

Operator definition can be found here :

[http://en.wikipedia.org/wiki/Operators\\_in\\_C\\_and\\_C%2B%2B](http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B)

Note that it is possible to use binary operator over variable of different fundamental type, this will imply a **promotion** of type of one of this two variable. For example :

```
double a;
int b;
double c = a + b;
```

a is automatically promoted to double for this operation.

# literals

You will probably need write some constant, like 10 or 42 or 3.1415. To do so C++ allow the following definition:

- **decimal integer**, for example: 0, 2, 63, 83
- **octal integer**, for example 0, 02, 077, 0123
- **hexadecimal integer**, for example 0x0, 0x2, 0x3f, 0x53
- **long integer**, for example 2L, 02L, 0x2L
- **unsigned integer**, for example 2U, 02U, 0x2U
- **unsigned long integer**, for example 2UL, 02UL, 0x2UL
- **float or double**, for example 0.0, 0.0f, 1.0e+10
- **char**, for example 'a', 'b'.

# void

`void` is used to specify :

that a function does not return a value

```
void foo (int);
```

that a function does not take an argument

```
int foo(void); // int foo ();
```

the base type for pointers to objects of unknown type

```
void* p;
```

# enumeration

Define integer constants

```
#include <iostream>
enum {ZERO, UN, DEUX};
enum NOMBRES {TROIS = 3, QUATRE = 4};
int main () {
    std::cout << UN
                << " " << DEUX
                << " " << QUATRE
                << std::endl;
}
```

## 02. Pointer, Array and string



# Pointer definition

A pointer type is a **fundamental type**.

For a type **T**, **T\*** is the type of the pointer to a variable of type **T**.

For example:

```
int * pointer_of_int; // is a pointer of int
float * pointer_of_float; // is a pointer of float
```

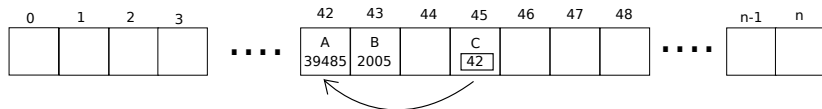
In an expression :

- the unary operator **&** give the **pointer of**
- the unary operator **\*** give the **object pointed by**

The implementation of pointers is directly bound to the **addressing mechanisms** of computers. For this reason pointers are also called address of variables.

# What are pointers actually?

To understand pointer we have to represent internal memory. The internal memory can be represented as a set of numbered box, each box can content an finite quantities of data :



This schema represent the possible memory layout of:

```
int A = 39485; // define A
int B = 2005;  // define B
int * C = &A;  // C is a pointer of int,
               // currently it point to A
int D = *C     // D = 39485;
```

# Examples of pointers

```
int main () {  
    char  c1 = 'a';  
    char * pc1 = &c1; // p1 hold the address of c1  
    char  c2 = *p1;   // c2 is 'a'  
  
    int zero = 0;  
    void * p = zero; // a pointer with value 0  
    void * p0 = 0;    // C++98 pointer with value 0  
    void * p1 = nullptr; // C++11 pointer with value 0  
}
```

# Arrays

For a type **T** and a constant positive integer **n**, **T[n]** is the type of **n** consecutive variables of type **T**. Each variables can be retrieve with their index from **0** to **n-1**.

```
char tab1 [12]; // array of 12 char.
float tab2 [24]; // array of 24 float.
int* ptab1 [6]; // array of 6 int pointer.
char tab2 [5] = {'a', 'b', 'c'}; // array of char.
char tab2p [] = {'a', 'b', 'c', 0, 0}; // same array as tab2
char tab3 [3] = {'a', 'b', 'c', 'd'}; // error
```

**n** must be constant, the following is forbidden:

```
int size = 10;
int tab[size]; // fail, size is variable
```

# Multidimensional array

```
#include <iostream>
using namespace std;

int main() { // definition of the main function
    int mat[3][4] = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            cout << mat[i][j] << " "; // NOT mat[i,j] !
        }
        cout << endl;
    }
    return 0;
}

// output:
// 0 1 2 3
// 4 5 6 7
// 8 9 10 11
```

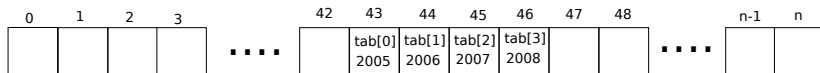
Note: Multidimensional array still being consecutive variables.

# Pointers and array

Array are managed as pointer, so the following array:

```
int tab[] = {2005, 2006, 2007, 2008};
```

can have the following layout:



and **tab** as the type of **int \*** and is equals to 43.

you can do the following things:

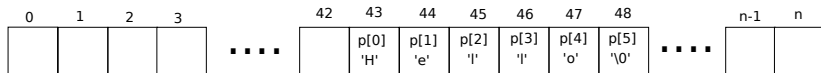
```
int * pointer_of_int = tab; // pointer_of_int is 43
int * pointer_of_third_element = &tab[2]; // this pointer is 45
int A = *pointer_of_int; // A equals 2005
```

# Pointers and string literals

String are array of char. By convention, the value of the last element of this array is 0, also called null character. For example :

```
char const s[] = "Hello";
```

The numbers of elements of s is 6 (5 characters + the null character). This array is **statically** allocated (definition later), which, in memory, look like :



# String and history

For backward compatibility you can assign it to a non const **char\***, as follow:

```
char* p = "Hello";
```

but the string literals should not be modified (static zone) and the result of such assignment is undefined ! The following code is wrong, even if it look fine:

```
int main () {  
    char* c = "toto";  
    c[0]='v';  
}  
g++ file.cc  
bash$ ./a.out  
Segmentation fault
```

If you want to **modify** it, **copy** it into an array as follow:

```
int main () {  
    char c[] = "toto";  
    c[0]='v';  
}
```



# Pointers aritmetics

- Size of 'box' are 8 bits (1 byte) in commun hardware
- An interger in commun hardware are 4 bytes, which mean they use 4 'boxes'.
- The sum and substration of interger with pointers are 'smart'.

```
int i_tab[10];
short s_tab[10];
int * i = i_tab;
int * s = s_tab;
/* i_tab[5] is equivalent to *(i + 5); */
/* s_tab[5] is equivalent to *(s + 5); */
/* but ((int)(i + 5) - (int)i) !=
      ((int)(s + 5) - (int)s) */
```

The value of pointer are updated according the pointed type. That mean incrementing a pointer of int will update the pointer to the integer just following the pointed int.

# 03. Statements

How to control the workflow

# Some definitions

```
/* expr: */
x = 5 + 2 * a;
/* blocks : */
{ }
/* branches: */
if (expr) then { } else { }
switch (expr) {
    case cst: statement; break;
    default: statement;
}
goto label; label:
continue;
break;
/* loops : */
while (expr) statement
for(expr; expr; expr) { }
do { } while (expr)
/* functions: */
void foo(args) { return; }
```

# Expression

My informal definition: an expression is the simplest statement, which do not have flow control in it, such as:

```
10 + 2
(a? b : a * 2) / foo(e) + *c
bar(a + 3 * b)
x = 3 * x + b
```

# Sequence and blocks

Expressions can be evaluated sequentially and grouped by blocks to create statements.

Sequential evaluation is done with the ';' operator: an expression followed by ';' is evaluated and its 'return' value is discarded.

All side-effects are completed before the next statement is executed

Sequence of expression can be grouped in a block with curly bracket: { } .

Previous expression can be turn to sequence, block and more as follow:

```
{  
    10 + 2;  
    {  
        (a? b : a * 2) / foo(e) + *c;  
        bar(a + 3 * b);  
    }  
    int x = 3 * x + b;  
    f(i);  
    int i = 10 + 2;  
}
```

## three **selection** statements

**if** (condition) block

**if** (condition) block **else** block

where **condition** is an expression evaluated and implicitly converted to bool.

**switch** (value) { case 0: statement ... default: statement }

where **value** is an expression implicitly evaluated and converted to an integer.

# if statements

```
if (cond0) {  
    if (cond1) {  
        /* some code */  
    } else {  
        /* some code */  
    }  
}
```

or

```
if (cond0) {  
    if (cond1) {  
        /* some code */  
    }  
} else if (cond2) {  
    /* some code */  
} else {  
    /* some code */  
}
```

# the **switch / case** statements

```
void f0 () {}  
void f1 () {}  
void fdefault () {}  
int main(int argc, char* argv[]) {  
    int val = 0;  
    switch (val) {  
        case 0:  
            f0(); break;  
        case 1:  
            f1(); break;  
        default:  
            fdefault(); break;  
    }  
    return 0;  
}
```

```
if (val == 0) {  
    f0();  
} else if (val == 1) {  
    f1();  
} else {  
    fdefault();  
}
```



# iteration statements

**while** (condition) statement

**do** statement **while** (condition) ;

**for** (for-init; for-cond; cont-expr) statement

```
for(int i = 0; i < 6; i++) { /*...*/ }  
for(int j = 0; j < 3; j++) { /*...*/ }  
int i = 0;  
for(; i < 12; i++) { /*... */ }  
for(;;i++) { /*...*/ }  
for(;;) { /*...*/ }
```

# the `while` Statement

```
while (T t = x ) foo();
```

is equivalent to

```
label :  
{  
    T t = x;  
    if (t) {  
        foo();  
        goto label;  
    }  
}
```

# the `for` Statement

```
for (for_init_statement ; condition ; expression ) {  
    statement ;  
}
```

is equivalent to

```
{  
    for_init_statement ;  
    while (condition) {  
        statement ;  
        expression ;  
    }  
}
```

no more **definition block** for variables in c++

**Define variables everywhere** (not only at the beginning of a scope as C)

**Define** an object as **close as possible** to its **point of use**

You can put the **for loop counter** inside the **for expression**

```
int * buf = new int[100];  
for (int i = 0; i < 100; i++) {  
    buf[i] = 0;  
}
```

# labeled statement

A statement can be **labeled**.

identifier : statement

**case** constant\_expression : statement

**default** : statement

the only use of an identifier label is as the target of an (infamous) **goto** -  
notice it exists and forget about it -

a label can be used in a goto before its definition

labels have their own name space and do not interfere with other identifiers

# Functions

Functions are used to group expr and statement together to reuse them easily. Functions look like:

```
int function_name (int args0, int args1,  
                  /*...,*/ int argsn)  
{  
    /*...*/  
    return something;  
}
```

Functions are called by passing them input and gather outputs, i.e. the return value. As folow:

```
y0 = call_function_0(x0, x1, x2);  
y1 = call_function_1(x0);
```

## 04. Declaration and Definition

# declaration versus definition

A **declaration** introduces a **name** with its **type**

```
void f (int) {} // definition of the function f
int i = 10;     // definition of the integer i
int main () {   // definition of the main function
    f(i);       // the compiler knows f and i
    return 0;
}
```

A **definition** allocates a **storage** for a **name**

- for a **variable** the compiler generates **space memory**
- for a **function** it generates **code** and ends up allocating **storage**

When **defined**, a **variable** and a **function** have an **address** (the address of the memory where they are stored)

You can take a **pointer** to a **variable** or a **function**



# forward declaration

```
void f (int);    // forward declaration of the function f
int i;           // definition of the integer i (with value 0)
int main () {    // definition of the main function
    f(i);        // the compiler knows f and i
    return 0;
}
```

In function 'main':

tutu.cpp:14: undefined reference to 'f(int)'

the compiler knows the **existence** of **f** but cannot find its **definition**

# the **extern** keyword

A **definition** is also a **declaration** when it is the **first time** the compiler meets the name

If you want to **declare** but not **define** a **variable** use **extern**

For a **function** **extern** is optional, there is no ambiguity between **function declaration** and **function definition** :

- **declaring** a **function** is giving the **prototype**
- **defining** a function is giving the **body**

Objects declared **const** and not **explicitly** declared **extern** have **internal linkage**

# forward declaration

```
void f (int);    // forward declaration of the function f
extern int i;    // declaration of the integer i
int main () {    // definition of the main function
    f(i);        // the compiler knows that f and i exist
    return 0;
}
```

In function 'main':

tutu.cpp:14: undefined reference to 'i'

tutu.cpp:14: undefined reference to 'f(int)'

the compiler knows the **existence** of **f** and **i** but cannot find their **definition**

**no** piece of **storage** has been allocated for **f** and **i**

## 05. Implement a sort algorithm

# Create a New C++ Project

- in the **eclipse** menubar
- choose the **New** submenu
- choose the **Project...**
- open the **C/C++** submenu and choose the **C++ Project**
- enter **Sorting** as the **Project Name**
- open the **Executable** submenu
- choose **Hello World Project**
- click **Finish**
- accept the C++ perspective

# create an array of integers

in the main function **create** an **array** of **integer** and **initialize** it

```
int main() {  
    cout << "my sort algorithm" << endl;  
    int tab[] = { 16, 50, 8, 3, 56, 23, 15 };  
    return 0;  
}
```

# create a sort and a print functions

**create** a **sort** function returning **void** and taking as arguments : the **size** of the array and a pointer to the **array**

```
void sort (int size, int* array) {}
```

**create** a **print** function returning **void** and taking as arguments : the **size** of the array and a pointer to the **array**

```
void print (int size, int* array) {}
```

call the **functions** in the **main**

```
int main() {  
    cout << "my sort algorithm" << endl;  
    int tab[] = { 16, 50, 8, 3, 56, 23, 15 };  
    sort(7, tab);  
    print(7, tab);  
    return 0;  
}
```

# implement a sort algorithm

## insertion sort :

- iterate on the array, find the smallest integer, exchange it with the first element of the array
- iterate a second time on the array, find the second smallest element and exchange it with the second element
- ... stop when you have placed all the elements

## bubble sort :

- iterate the array by comparing adjacent elements
- exchange each pair of adjacent elements which are out of order
- the first pass puts the largest element in the last case of the array
- the second pass puts the second largest elements into position
- after the  $k^{st}$  pass, the sub-array  $N - k, \dots, N - 1$  is ordered
- the array is sorted when no exchange are required during a pass



# the print function

```
void print (int size, int* array) {  
    cout << "[ ";  
    for (int i = 0; i < size; i++) {  
        cout << array[i] << " ";  
    }  
    cout << "];"  
}
```

# the bubble sort function



# the insertion sort function

```
void sort(int size, int* array) {  
    if (size > 0) {  
        int smallest;  
        int smallest_indice;  
        for (int i = 0; i < size; i++) {  
            smallest_indice = i;  
            smallest = array[smallest_indice];  
            for (int j = i; j < size; j++) {  
                if (array[j] < smallest) {  
                    smallest = array[j];  
                    smallest_indice = j;  
                }  
            }  
            cout << array[smallest_indice] << endl;  
            swap(array, i, smallest_indice);  
        }  
    }  
}
```

## 06. Storage Duration

# Static Store, Execution Stack and Heap

## Static objects :

- **stored** on the **static store**
- **allocated before** the **main** begins
- **de-allocated after** the **main** terminates

## Automatic objects :

- **stored** on the **execution stack**
- **allocated** when the program **meets** their **definition**
- **de-allocated** at the **end** of the **life time** (scope)

## Dynamic objects :

- **stored** on the **heap**
- **allocated** at the **call** of **new**
- **de-allocated** at the **call** of **delete**

# Exemple of static object

Static object are declared as global or as **static**.

They live for the entire duration of the programme, and they cannot be freed or deleted.

For exemple:

```
const int i = 10;
int j;
int my_static_array[20];
double my_static_double;
void foo() {
    static int my_static_int_in_foo = 30;
}
```

# Exemple of automatic object

Automatic object live inside functions or blocks.

They are automaticly freed or deleted at the end of there life.

```
void foo () {  
    short my_automatic_short; // leave until end of foo  
    /* do some thing */  
    bar();  
}  
  
double bar () {  
    double my_automatic_double; // live until end of bar  
    /* do something */  
    return my_automatic_double;  
}  
  
int main() {  
    int my_automatic_int = 15; // live until end of main  
    foo();  
    return 0;  
}
```

# Exemple of dynamic storage

```
int * foo (int size) {  
    // dynamic allocation  
    int * my_dynamic_array_of_int = new int[size];  
    return my_dynamic_array_of_int  
}  
  
int bar() {  
    int i = 10;  
    /* do something */  
    return i;  
}  
  
int main() {  
    int * my_array = foo();  
    // dynamic allocation  
    int * tab = new int [bar()];  
    return 0;  
}
```

we allocate arrays whose size is unknown at compile time using dynamic allocation

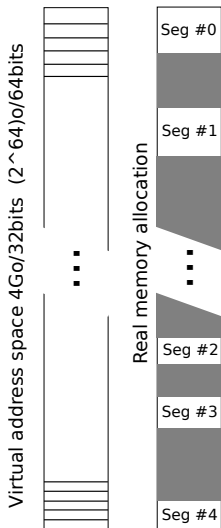


# How executable are executed ?

Executables are not executable AS-IS, kernel launch executable from a specially formatted file. on unix elf32/elf64 format is used. Step to launch an executable:

- 1 Executable file is loaded and interpreted.
- 2 an executable context is created with a virtual memory allocation
- 3 librarie file are loaded, interpreted and put into the virtual address space of the executable
- 4 the kernel API is mapped into the virtual address space.
- 5 kernel launch the executable context and call the **main** function

# Executable work principles



## Content of segments:

- read-only data sections : static const characters string, static const variables
- read-only executable sections : functions content, libraries contents, kernel API
- read-write data sections : program and libraries allocated data, execution stack
- read-write executable (uncommon) : some code are generated on the fly (for exemple emulator can translate original assembly to native assembly, some mutable viruses). C++ does not generate mutable code, executable section are always read only.

## the `new` and `delete` operators

In C++, for a type `T` and `p` of type `T*`, the **dynamic memory allocation** is done with:

- `new T` , allocation of one `T`
- `new T[n]` , allocation of array of `n T`
- `delete p` , delete one `T`
- and `delete[] p` delete an array of `T`

```
int* pi = new int;  
double * my_array_of_float = new float[42];  
delete pi;  
delete[] my_array_of_float;
```

`new` and `delete` are used as **malloc** and **free** but are not equivalent. `new` and `delete` call constructor and destructor when **malloc** and **free** do not. (We see later constructor and destructor)

## 6.1. Functions Part 2

# Function arguments

Function arguments, by default, are passed by copy.

```
void foo (int i) {  
    // an integer variable i is automatically allocated in the  
    // execution stack and initialized to the value of the argument  
    i = i + 12;  
}  
  
int main () {  
    int j = 1; // an integer variable j is automatically allocated  
               // in the execution stack and initialized to 1  
    foo(j);    // the value of j is copied in the variable i of foo  
               // i is automatically deallocated from the stack  
               // j is still equal to 1  
}
```

# Usage of pointers

pointer arguments are also passed by copy

```
void foo (int* i) {  
    // a pointer to an integer variable i is automatically allocated  
    // in the execution stack and initialized to the value of the  
    // argument i.e. the address of i  
    *i = *i + 12;  
    // the value pointed by i is changed  
}  
  
int main () {  
    int j = 1; // an integer variable j is automatically allocated  
               // in the execution stack and initialized to 1  
    foo(&j);   // the address of j is copied in the variable i of  
               // foo i is automatically deallocated from the stack  
               // j now is equal to 13  
}
```

# Recursive function

## Functions can call themselves

```
#include <iostream>
using namespace std;
void foo() {
    cout << "calling foo" << endl;
    foo();
}

int main() {
    foo();
}
```

But there must be a condition to stop !

# factorial function

```
#include <iostream>
using namespace std;
double fact (double n) {
    if (n <= 1) return 1;
    return n*fact(n-1);
}

int main() {
    cout << fact(12) << endl;
}
```



# Are recursive functions always the good way to code ?

```
int fibonacci(int n) {  
    if (1 == n || 2 == n) {  
        return 1;  
    } else {  
        return fibonacci(n-1) + fibonacci(n-2);  
    }  
}  
  
int main() {  
    for (int i = 1; i <= 10; i++) {  
        cout << fibonacci(i) << endl;  
    }  
    return 0;  
}
```

What do you think of this implementation of fibonacci ?

# Recursive function drawback

Recursive function allow write some algorithm quickly, but are often less permorment and consume more memory than there iterative equivalent.

```
int fibonacci(int n) {  
    int u_n_minus_2 = 0;  
    int u_n_minus_1 = 1;  
    int fib;  
  
    for (int i = 2; i <= n; i++) {  
        fib = u_n_minus_2 + u_n_minus_1;  
        u_n_minus_2 = u_n_minus_1;  
        u_n_minus_1 = fib;  
    }  
    return fib;  
}
```

The iterative version of fibonacci is a linear time algorithm, the recursive one is an exponential time algorithm.

# Function **overloading**

C++ being strongly typed, it can deduce the function to call from its arguments type : this is function **overloading**

With **overloading**, the same function name can have different behaviour depending on context :

```
void print(int);  
void print(float);  
void print(Date);  
void print(IntStack*);
```

Note: the return type cannot be used to deduce the function to call. You cannot overload a function as follow:

```
void foo(int);  
int  foo(int);
```

but you can, even if it is not recommended:

```
void foo(int);  
int  foo(float);
```

# Common pitfal of dynamic object

The following program will compile (with warning probably).

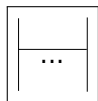
```
double * bar () {  
    double my_automatic_double = 42; // live until end of bar  
    /* do something */  
    return &my_automatic_double; // could warn about this line  
}  
  
int main() {  
    int my_automatic_int = 15; // live until end of main  
    double * my_automatic_pointer = bar();  
    int my_cofe = 0xCAFE;  
    std::count << *my_automatic_pointer << std::endl;  
    return 0;  
}
```

But it behaviour is undefined and can segfault.

# Function call principles (1/2)

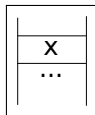
```
int x;  
x = foo(a,b);
```

1

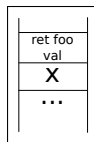


the stack

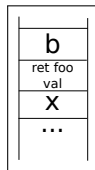
2

execute  
"int x"

3

allocate  
space for  
return  
value

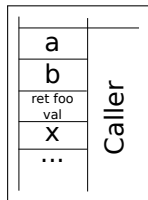
4

execute  
"b"

# Function call principles (2/2)

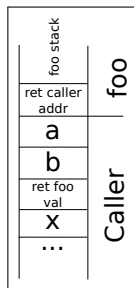
```
int x;
x = foo(a,b);
```

5



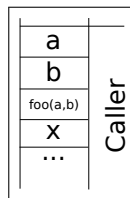
execute  
"a"

6



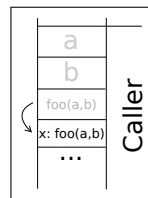
execute  
"foo"

7



store  
return  
value in  
right  
location

8



execute  
"x ="  
and  
cleanup

## 07. Program (2)

# header and implementation files

having **all** your **code** in a **single** file is **hard**

Best Practice™ recommend to **organize** a **program** into **files**

**header** files (**sort.h**) will contain **interface informations** (declarations)  
you need almost **everywhere**

```
#include <standard_include> // from standard include directory  
#include "personal_include" // from current directory
```

**implementation** files (**sort.cpp**) will contain **executable code** and **data definitions**

**header** files will be **included** (**#include "sort.h"**) by **implementation files** when needed



# implementing a header file for your **sort algorithm**

you have **implemented** an **algorithm** (sort algorithm)

The set of functions which are accessible outside .c file are called **interface functions**

interface functions must be declared in a **header file**

For the sort algorithm this file must be named **sort.h**

```
#ifndef SORT_H_
#define SORT_H_
void print (int* array, int size);
void sort (int* array, int size);
#endif /* SORT_H_ */
```

Notice the use of **pre-processeur** global variable **SORT\_H\_** to avoid **multiple inclusions**

# implementing the **source file** of your **sort algorithm**

**implementation** of interface functions are in a **source file** By convention :

- it has the same prefix name as the header file but ends by **.cpp**  
**sort.cpp**
- it includes the header file

```
#include<iostream>
/* include header */
#include "sort.h"

using namespace std;

void sort (int* array, int size) {
    /* some code */
}

void swap (int* array, int pos1, int pos2) {
    /* some code */
}

void print (int* array, int size) {
    /* some code */
}
```

# using your header file

To use your header file, you must include it in an implementation file, for example a **main.cpp** file

```
#include "sort.h"
int main () {
    int tab [9] = {12,4,5,1,78,456,999,103,0};
    sort(9, tab);
    print(9, tab);
}
```

# eclipse: Split you sort algorithm in the three files(1)

click **left** on your **sort Project**, choose **New** then **Header File**

Enter the **sort.h** name in **Header File** :

Choose the **Template Default C++ header template**

click **Finish**

You obtain :

```
#ifndef SORT_H_
#define SORT_H_

#endif /* SORT_H_ */
```

## eclipse: Split you sort algorithm in the three files(2)

**Enter** the **interface functions prototypes** in the **sort.h** file

Create an implementation file named **main.cpp** :

- **left click** on **Project** name
- choose **New** then **Source File**
- enter the name **main.cpp**
- choose **Template Default C++ source template**
- click **Finish**

include the **sort.h** file and the **main** function

**Run** it

# What is **separate** compilation

every **source file** can be **compiled** in an **object code**

On **Linux platforms**, the **object code** is in **.o** files

**object code** is **machine dependant code**

an object code file cannot be run

It requires a **further Link stage** where a **Linker program** :

- reads the object code
- assembles it into an executable file
- and writes that file to your disk

# Doing **separate** compilation on your sort project **without** **eclipse(1)**

You have **three** files : **sort.h**, **sort.cpp** and **main.cpp**

the **header** file **sort.h** being included in the two other files, will never be **compiled alone**

the **two source files** must be **compiled** all by themselves

```
g++ -c sort.cpp  
g++ -c main.cpp
```

- 1 **sort.cpp** is **compiled**, not **linked**  $\Rightarrow$  **sort.o**
- 2 **main.cpp** is **compiled**, not **linked**  $\Rightarrow$  **main.o**

**Notice** the **-c** in the **command line** **g++ -c** to choose the **compilation option**

# Doing **separate** compilation on your sort project **without eclipse**(2)

Then you **link** the **.o** files to **create** an **execution file**

```
g++ sort.o main.o
```

By default on **Linux** the **executable file** is **a.out**

What is the link doing ?

- in the **object file main.o**, the **sort** and **print** functions are :
  - **declared** from the sort.h file previously included
  - called
  - but not defined
- in the **object file sort.o**, the **sort** and **print** functions are :
  - **declared** from the sort.h file previously included
  - **defined** (every function has a body)
  - but not **called**
- *they were meant to meet*



# Doing **separate** compilation on your sort project **without** **eclipse**(3)

You can give you executable file a name with the `-o g++` command line option

```
g++ -o mysort sort.o main.o
```

you can do all of this in one pass

```
g++ -o mysort sort.cpp main.cpp
```

**sort.cpp** and **main.cpp** are compiled and linked to produce the **mysort** binary

**sort.h** is **included** before the compilation by the **pre-processor** (`#` directives) in **sort.cpp** and **main.cpp**

# multiple definitions

there must be a **unique** definition of a type (class, template, ...)

c++ accepts **multiple definitions** if :

- they appear in **different** compilation units (file, ...)
- they are **token-for-token identical**
- in fact they must "come" from a **common** source file

## multiple inclusions of **header** file

to avoid **multiple** definitions in the **same** compilation unit, use **pre-processing** directives :

in **sort.h**

```
#ifndef SORT_H
#define SORT_H
void print (int* array, int size);
void sort (int* array, int size);
#endif // SORT_H
```

the code between **ifndef** and **endif** will be ignored once the global variable **SORT\_H** is **defined**

# Creating a library

```
g++ -L/path/to/Library/directory -l/path/to/library/name  
-I/path/to/includes ...
```

# Header files (\*.h files).

- named namespaces
- type definitions
- template declarations
- template definitions
- function declarations
- inline function definitions
- data declarations
- constant definitions
- enumerations
- name declarations
- include directives
- macro definitions
- conditional compilation directives
- comments
- ...

# Implementation files (\*.cxx files).

- ordinary function definitions
- data definitions
- unnamed namespaces
- comments
- ...

# conditional inclusion

a preprocessing directive begins by the `#` token

```
#if VERSION == 1
```

check whether the identifier `VERSION` evaluates to the value 1

```
#define INCFILE "version1"
```

`INCFILE` is defined as a macro name with value `"version1"`

```
#elif VERSION == 2
```

else check whether `VERSION` evaluates to 2

```
#define INCFILE "version2"  
#else  
#define INCFILE "versionN.h"  
#endif  
#include INCFILE
```

`include` causes the replacement of that directive by the entire contents of the source file identified by the macro name `INCFILE`

# include header

```
#include <toto>
```

searches for a header identified between the < and > delimiters

causes the replacement of that directive by the entire contents of the header

how the places are specified or the header identified is implementation-defined



## predefined macro names

The following macro names are defined by the implementation :

- `__LINE__` is the line number of the current source line
- `__FILE__` the presumed name of the source file
- `__DATE__` the date of translation of the source file
- `__TIME__` the time of translation of the source file

```
int main () {  
    cout << __LINE__ << " " << __FILE__ << " " ;  
    cout << __DATE__ << " " << __TIME__ << endl;  
}  
2 test.cpp Mar 11 2005 16:34:46
```

## 09. Data Structure

# Adding new structured types to your programs

When you want to **group** a **set of types** to **compose** a **new type**, use the **struct** `{}` keyword

For example, i need a **Date type** with three integer **fields** **day**, **month** and **year** :

```
struct Date {  
    int day, month, year;  
};
```

**Notice** the `;` at the end of the **class definition** : Try to never forgot it !

# Procedural Programming Style

Suppose you want to write **functions** that work on this **new data type** :  
**Date**

```
struct Date {  
    int day, month, year;  
};  
  
void Tomorrow (Date* date) { /* some code */ }  
void PrintDate(Date* date) { /* some code */ }  
void InitDate (Date* date, int d, int m, int y) {  
    date->day = d;  
    date->month = m;  
    date->year = y;  
}
```

This is the way **procedural languages** (such as C) describe a **set of same-typed objects** sharing **characteristics** and **behaviours** :

- **structure of data members**
- **plus** a set of **global functions** passing a structure's pointer as argument

# Procedural Programming is a poor programming style

**Problem:** the lack of **explicit connection** between

- the **structure** (Date)
- **and** the **functions manipulating the structure** ( InitDate, Tomorrow and PrintDate)

## declaring **member functions**

When **functions** operate on a **particular structure**, **declare** those functions **inside** the structure **definition**

they become **member functions**

```
struct Date {  
    int day;  
    int month;  
    int year;  
    void Init (int, int, int);  
    void Tomorrow ();  
    void Print (); // on standard output  
};
```

the C++ compiler turns the name Date into a **new type name** of the language (no need of **typedef** like in C)

# comments

in C++ a **comment** begins by `//` and goes to the **end** of the **line**

in C++ C comments `/* */` are still available

# member functions

The functions **first argument** from the C-style version **vanishes** in the **member functions**

In C++ you are not **forced** to **pass** the address of the class as an argument : the compiler **does it for you** as soon as you **declare** the function inside the **structure**

you can **define** a member function **outside** the scope of the **structure** : you simply have to specify to **what structure** the **function** belongs



# full specification

C++ has a new operator `::` named the scope resolution :

```
void Date::Init (int d, int m, int y) {  
    day = d; // no more date->  
    month = m;  
    year = y;  
}
```

In **member** functions, the **member selection vanishes** : member **names** are used **without explicit** reference to an **object**

a **member function** knows the **hidden argument** and will **apply it** the **member selector** whenever you refer to one **member**

**Data member names** and **member function names** do not collide with same names inside any **other structure**

# user-defined type

a **structure** is a **user-defined type**

the **fundamental** idea is to **separate** :

- the **interface** : a set of data and functions you offer to the outside to manipulate the structure objects
- from the **implementation** : the details you may wish to change

```
struct Date {  
    int day, month, year;  
    void Init (int, int, int);  
    void Tomorrow ();  
    void Print ();  
};
```

# creation of Date objects

Date is a **new type** : you define a Date **object** like a built-in type

```
Date d;  
Date today = {15, 3, 2004};  
  
int f (Date* date) {  
    Date today;  
    today.Init(15, 3, 2004);  
    today.day = date->day;  
    date->Print();  
}
```

You can access data members and member functions

**type checking** is performed with **user-defined types** as with **built-in types**

C++ will check that the **function** `f` is called with a `Date` pointer as its first argument, ...

# creation of Objects

like built-in types, objects of user-defined type can be create in the three zones :

- staticDate in the **static store**
- stackDate in the **execution stack**
- newDate in the **heap**

```
Date staticDate;  
int foo () {  
    Date stackDate;  
}  
int bar () {  
    Date* newDate = new Date;  
}
```

# The implicit argument **this**

In a member function, you refer directly to members (data or function) by their names. Such a code is more compact: easier to write and to read.

The address of the structure used to call a member function is called **this** and it can be used in member functions, as follow:

```
void Date::Init (int day, int month, int year)
{
    this->day = day;
    this->month = month;
    this->year = year;
}
```

# The **this** pointer exemple

**this** is initialized to the addresse of the object throught which the member function is called

```
Date* today = new Date;  
today->Init(15, 03, 2004);
```

In the function we get:

```
Date::Init (int d, int m, int y) {  
    // Date * this is today;  
    this->day = d;           // day    is today->day  
    this->month = m;        // month  is today->month  
    this->year = y;         // year   is today->year  
}
```

# Data Access Control

# Control access to structure's fields

The first reason to control the access to members variable or member function of a structure is to avoid programmer mistakes.

If a **user** is able to **access every thing**, he will normally end by doing wrong things

```
void Date::Init (int d, int m, int y) {  
    if (DayIsValid(d, m, y)) day = d;    // You do nice verifications  
    if (MonthIsValid(d, m, y)) month = m; // to keep your date correct  
    if (YearIsValid(d, m, y)) year = y;  
}  
  
void f (date* date) {  
    date->day = 120;    // stupid, you must prevent a user to do it.  
}
```



# hide implementation details

The second reason is to **forbid** to the **outside** access to **members** you want to **keep private** to your **implementation**

Such **members** may **change** and it must **not affect** the **outside** code

Separate **public interface** from **implementation details**

C++ introduce **keywords** to deal with **access control: public, private** and **protected**

Called **access specifiers**, they are used **only** in **structure declaration**

# c++ access control :public , private and protected members

```
struct Date {  
private:  
    int day, month, year;  
public:  
    void Init (int, int, int);  
    void Tomorrow();  
    void Print();  
};
```

**public:** member declarations that follow are **available** to **everyone**

**private:** member declarations that follow are available **only** in **function members** of that type

**protected** acts like **private** (for **inheritance** purpose)

## example of c++ access control

Trying to **access private members outside member functions** raises a **compile-time error**

```
void f (date* date) {  
    date->day = 120;  
}
```

file.C:10: member 'day' is a private member of class 'Date'

and the compilation is **aborted**

# restricted access

you **restrict** the access to **data members** to an **explicitly declared** list of **functions** (the public interface)

**data members** can take **illegal** values only inside **member functions** :  
the **debugging** is **easier**

if you **change** the **internal representation** of a **structure**, you need only change the **member functions** (as the data members cannot be involved in other functions) : the **code** is more **extensible**

as **user code depends only** on the **public interface** : it needs **not** be **rewritten**

# friend

to **allow** an outside **function** to access the **private** and the **protected** members of a structure : **declare it friend** to the structure

**like you do, a structure chooses its friends**

```
struct Date {  
    friend void foo (Date*);  
    .../...  
};  
void foo (Date* m) {  
    m->day = 120;  
}
```

the structure Date gives access to its **private** and **protected** fields to the **global** function void foo (Date\* m)

# friend

you can declare **friend** : **global function**, **member function** of **another structure**, a **whole structure**

```
struct Date;
struct BirthDay {
    void Print (Date*);
};
struct Date {
    friend void foo (Date*);
    friend void BirthDay::Print (Date*) ;
    friend struct Calendar;
};
```

A **friend declaration** is also a **declaration** : Calendar and foo do not need to be previously declared but BirthDay needs to be

## the `class`

In a `struct`, members are public by **default**

if you **forget** to restrict the access to a member that was intended to implementation details :

- you introduce a potential source of **bugs** because the outside can change it
- if users involve it inside their code : you cannot change your internal implementation any more (or the user's code becomes obsolete)

In a **`class`** members are private **by default**

If you **forget** to give access to a member that was intended to the public interface : you simply get a **compile-time error**

Use **`class`** rather than **`struct`**

`struct` is **kept** in C++ for **backward** compatibility with C

# 11. TP2 : Implement a stack of integers



# Create a New C++ Project

- in the **eclipse** menubar
- choose the **New** submenu
- choose the **Project...**
- open the **C/C++** submenu and choose the **C++ Project**
- enter **intstack** as the **Project Name**
- open the **Executable** submenu
- choose **Hello World Project**
- click **Finish**
- accept the C++ perspective

# Create an intstack class

- **left click** on the **project** name
- choose **New** then **class**
- enter the class name **IntStack**
- **refuse** the two **Methods Stubs** : **constructor** and **destructor**
- click **Finish**

You obtain :

```
/**
 * IntStack.h
 */
#ifndef INTSTACK_H_
#define INTSTACK_H_

class IntStack {
};

#endif /* INTSTACK_H_ */
```

# What is a stack ?

a **stack** is a **container** in **which** :

- the **last inserted element**
- will be the **first outputed element**
- last in first out

# What must be the behaviour of your stack ?

the **behaviour** of our **stack** is :

- you can **push** an integer value (lets say **1**), the **stack stores** it
- there is a **limited number** of **integer** that can be **stored**
- you can **test** if the stack is **empty**
- you can **test** if the stack is **full**
- you can **push** another integer value (lets say **2**), the **stack stores** it
- you can **pop** the **stack**, this function will return the integer **2** to its caller
- you can **pop** the **stack** another time, the function will return the integer **1** to its caller
- if you try to **pop** the **stack** another time, the program will answer that the stack is **empty**
- you cannot **push** an **integer** inside a **full** stack

# What are the members of your structure ?

- an integer array to store pushed integers
- an indice to remember the top of the stack (where the last interger has been stored)
- the size of the stack
- a function to create an array of the given size
- the functions pop, push, empty, full, print, ...
- ...
- members are private or public
- You must have three files :
  - intstack.h with the class definition containing the functions declarations
  - intstack.cpp with the class functions definitions
  - main.cpp that includes intstach.h and uses it

## 12. Constructor

```
class IntStack {
    int* stack;
    int size, top;
public:
    void Initialize () {
        size = 100;
        stack = new int [size];
        top = 0;
    }
    void push (int i) {
        if (!full()) {
            stack[top] = i;
            top +=1;
        }
    }
    int pop () { if (!empty()) return stack[--top]; }
    bool empty () { return top == 0; }
    bool full () { return top == size; }
};

int main () {
    IntStack myStack;
    myStack.push(120);
}
```

What happens in this code?

This program ends up with a **Segmentation fault (core dumped)** problem



# initialization

**problems occur** when the programmer **forgets** to **initialize members**

the use of **functions** such as `IntStack::Initialize` to provide **members initialization** is **error prone**

the more often a user does not know **how to initialize** an object and even that **he must initialize** it

In C++ **initialization** was considered to be **too important** to be **let** to the **user**

The programmer can **guarantee initialization** of **every object** by providing a special function called a **constructor**

# initialization

When a **class** has a **constructor** :

- the compiler automatically **calls** that **constructor** when an object is created
- all objects of that class will be initialized by a constructor call

As the **compiler** is responsible for calling the **constructor**, it must know the **destructor name**

The **name** of the **constructor** is the **name** of the **class**

A **constructor** can take arguments

A **constructor** cannot return a value

```
class IntStack {
    int* stack;
    int size, top;
public:
    IntStack () { // constructor definition
        size = 100;
        stack = new int [size];
        top = 0;
    }
    void push (int i) {
        if (!full()) {
            stack[top] = i; top +=1;
        } }
    int pop () { if (!empty()) return stack[--top]; }
    bool empty () { return top == 0; }
    bool full () { return top == size; }
};

int main () {
    IntStack myStack;
    // calls IntStack::IntStack()
    myStack.push(1);
}
```

The **constructor** is called when an object of type IntStack is **created**

## add a **constructor** to Date

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
public:  
    void Init (int, int, int);  
    void Tomorrow ();  
    void Print ();  
};  
int main () {  
    Date d;  
    Date date;  
    date.Init(15, 3, 2004);  
}
```

this **code compile** properly

```
class Date {
private:
    int day;
    int month;
    int year;
public:
    Date (int, int, int);
    void Tomorrow ();
    void Print (); // on standard output
};

int main () {
    Date d;
    Date date (15, 3, 2004);
}

In function 'int main()':
error: no matching function for call to 'Date::Date
error: candidates are: Date::Date(const Date&)
error:           Date::Date(int, int, int)
```

## default constructor

**Default constructor** is the **constructor without argument**

If you do not provide a **constructor** for a class (struct), the **compiler** will **implicitly generate one** for you

But as soon as you **define a constructor**, C++ **stops defining default constructor**

# Be Careful

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
public:  
    Date (int, int, int);  
    void Tomorrow ();  
    void Print (); // on standard output  
};  
int main () {  
    Date d (); // no problem  
    Date date (15, 3, 2004);  
}
```

**Date d()** is function declaration (no argument, returning a Date object)

# initialization of **array** and **default** constructors

```
Date Cal [] = {Date(15, 3, 2004), Date (16, 3, 2004)};  
Date Bad [3] = {Date(21, 10, 1998)}; // ERROR no default constructors for Date
```

**Default constructor** allows you to **initialize every object in an array**

If you add a **default constructor** to Date, you can write :

```
Date DAT [100]; // calls 100 times the default constructor
```



# function overloading for **constructor**

to **initialize** an object in **more than one way** : use **overloading**

```
IntStack::IntStack () {  
    size = 100;  
    stack = new int [size];  
    top = 0;  
}  
IntStack::IntStack (int s) {  
    size = s;  
    stack = new int [size];  
    top = 0;  
}
```

# multiple constructors

```
class Date {  
private:  
    int day, month, year;  
public:  
    Date (int, int, int);  
    Date (int, int); // today year  
    Date (int); // today year and month  
    Date (); // today  
};
```

to **reduce** the **number** of **constructors** use **default arguments**

it allows you to **call** the **same function** (global or member) in **different ways**

## constructor **default** arguments

**Default arguments** are placed in the **declaration** not in the **definition** of functions

```
extern int todayDay, todayMonth, todayYear;  
class Date {  
private:  
    int day, month, year;  
public:  
    Date (int = todayDay, int = todayMonth, int = todayYear);  
};
```

A **default** argument **cannot** be followed by a **non-default** argument

When you **start** using **default** argument, the **remaining** arguments must be **defaulted**

# constructor default arguments

```
class IntStack {  
    IntStack (int = 100);  
};  
IntStack::IntStack (int s) {  
    size = s; stack = new int [size]; top = 0;  
}
```

# the `new` operator

You call **new** :

The **compiler** allocates **storage** to hold the object

If **allocation** successes it **invoke** the **constructor** to initialize that storage

```
int main () {  
    Date* aDate = new Date (15, 3, 2004);  
    Date* defaultDate = aDate;  
    aDate->Tomorrow();  
    delete defaultDate;  
    aDate->Tomorrow(); // Welcome to memory fault !  
    delete aDate; // undefined : serious run-time error !  
    Date* tab = new Date[100];  
    delete tab; // undefined : serious run-time error !  
}
```

## conclusion about new and delete

Because `malloc` and `free` don't know about **constructors** (and destructors) : use `new` and `delete` rather than `malloc` and `free`

are undefined :

- to call `free` on a pointer allocated by `new`
- to call `delete` on a pointer allocated by `malloc`
- to use `delete` on a pointer that has **already** been **deleted**
- to use `delete[]` on a pointer allocated by `new`
- to use `delete` on a pointer allocated by `new[]`

```
class IntStack {  
    int* stack;  
    int size;  
    int top;  
public:  
    IntStack (int s = 100) {  
        size = s;  
        stack = new int [size];  
        top = 0;  
    }  
};  
void foo () {  
    IntStack myStack;  
}
```

What happens after a call of foo() ?

# memory **leak**

**after returning** from the function `foo`

the pointer `myStack->stack` is lost

you cannot **delete** the **dynamic memory** you have **allocated** in the **constructor**

**not deleting** an object is **not** an **error** as far as the language is concerned : it is only a **waste** of **space** (memory leak)

but if your **program** is meant to **run** for a **long time**, such memory leaks become a **serious problem** (swap)

you need a **way** to **delete** the **dynamic memory** allocated in **constructor** : a **destructor**



# the **destructor**

**Cleanup** is as **important** as **initialization** : C++ provides a **destructor** and **guarantees cleanup**

**Syntax** for the **destructor** is **similar** to the syntax for the **constructor** but **prefixed** by a  $\sim$

The **destructor** is **called automatically** when an automatic object **goes out of scope**

## a destructor for IntStack

```
class IntStack {
    int* stack;
    int size;
    int top;
public:
    IntStack (int);
    ~IntStack ();
};

IntStack::IntStack (int s = 100) {
    size = s;
    stack = new int [size];
    top = 0;
}
```

```
IntStack::~~IntStack () {
    delete [] stack;
    // [] because you delete an array
}

int foo (int size) {
    IntStack myStack (size);
}
```

the **dynamically allocated** array `myStack->stack` is deleted **when returning** from the function `foo`

# a destructor for Date ?

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
public:  
    Date (int, int, int);  
    void Tomorrow ();  
    void Print ();  
};  
void foo () {  
    Date date (15, 3, 2004);  
}
```

Does the class Date need a destructor ?

## when do you **need** destructor ?

You don't need a **destructor** for Date : **nothing special** is done in the Date constructor (just initialization of integers)

You **need** a **destructor** when you do something **special** in a **constructor** :

**dynamic memory allocation**

**file manipulation** : you open a file inside the constructor

...

⇒ i. e. **something that needs to be end properly,**

```
#include "Date.H"
#include "IntStack.H"
extern int day;
extern int month;
int year;
Date today (day, month, year);
IntStack intStack;
extern void f ();
int main () {
    Date date;
    Date* anotherDate = new Date (1,12,1998);
    { IntStack aStack;
      aStack.Push(0);
    }
    delete anotherDate;
}
```

declaration ? definition ? life time ? constructor ? destructor ? static  
store ? execution stack ? heap ?

# declaration/implementation files

**Use access control** : separate **public** interface from **private** internal details

**Use separate compiling** : break your code into **different files** (translation units) that you can **compile separately** and then **link together** to obtain a **binary**

**Part declaration from implementation** : put declarations in an header file (file.h or file.H), put the implementation in another file (file.C, file.c, file.cpp, file.cxx ...)

The **header file** is the one you **include** in the other files

## restriction to the c++ implementation hiding

A **class** declaration must contain **every member** declarations

You cannot **hide** the **private part** (implementation dependant) of your class, you can avoid the user to **use it** but **not to read it**

Every time you touch an header file (to change your implementation) you must recompile files that depend on that header file (that include it)

```
// file IntStack.h
#ifndef INTSTACK_H
#define INTSTACK_H
#include <iostream.h>
class IntStack {
    friend std::ostream& operator<< (std::ostream& os, const IntStack& s);
    int* stack;
    int size;
    int top;
public:
    IntStack (in s = 100) { stack = new int [size = s]; top = 0;}
    void push (int i) { if (!full()) stack[top++] = i; }
    int pop () { if (!empty()) return stack[--top]; }
    bool empty () const { return top == 0; }
    bool full () const { return top == size; }
};
#endif // INTSTACK_H
```



# IntStack.cxx

```
// file IntStack.cxx
#include "IntStack.h"
std::ostream& operator<< (std::ostream& os,
                          const IntStack& s) {
    if (s.empty()) { os << "empty"; }
    for (int i = 0; i < s.top; i++) {
        os << s.stack[i] << " ";
    }
    os << std::endl;
}
```

```
in main.cxx
int main () {
    IntStack myStack;
    std::cout << myStack;
    myStack.push(1);
    myStack.push(2);
    myStack.push(3);
    std::cout << myStack;
}
```

# 13. TP2 : Continuing the implementation of your integers stack (1)

- add a constructor
- add a destructor

# 14. Reference

# reference in c++

a **reference** is like a **constant pointer** that is **automatically dereferenced**

It is **usually used** for **function arguments** list and **return value**

When a **reference** is **created** it must be **initialized** to an **existing object**

```
int i = 12;  
int& ri = i;  
ri++;    // i is now 13  
ri=156;  // i is now 156
```

**Incrementing** `ri` here is **incrementing** `i`

# reference **rules**

A **reference** must be **initialized** when it is **created**

When **created** a **reference cannot change** to **refer** to **another** object

You **cannot have null reference**: a **reference** is **connected** to an **existing piece of storage**

# reference used in **functions**

When **references** are **used** for **function arguments**, any **modification** through the **reference inside** the **function**, cause **change outside** the function

```
void swapP (int* pi, int* pj) {  
    int aux = *pi; *pi = *pj; *pj = aux;  
}  
  
void swapR (int& i, int& j) {  
    int aux = i; i = j; j = aux;  
}  
  
int main () {  
    int a = 12;  
    int b = 89;  
    // a == 12, b == 89  
    swapP(&a, &b);  
    // a == 89, b == 12  
    swapR(a, b);  
    // a == 12, b == 89  
}
```

# reference and **lvalue**

the **initializer** for a T& **must be** a **lvalue** of type T

```
int& ri = 1;  
ex20.cxx: In function 'int main()':  
ex20.cxx:2: error: could not convert '1' to 'int&'
```



## reference on **const** object

the **initializer** for a `const T&` **need not** be a **lvalue** of type `T`

```
const int& ri = 1;
```

a **temporary object** is **created** with the **value**

```
const int temp = 1;  
const int& ri = temp;
```

the **temporary** is **used** as the **initializer** for the **reference**

the **temporary persists** until the **end** of its **reference scope**

## reference on **return** values

with **reference**, **functions** can be **used** on both **left-hand** and **right-hand** side of an **assignment**

```
class Foo {  
    int value;  
public:  
    Foo (int i) { value = i; }  
    int& getValue () {  
        return value;  
    }  
};  
int main () {  
    Foo foo(10);  
    foo.getValue()++;  
    int i1 = foo.getValue();  
    // i1 == 11  
}
```

# reference to **function**

```
void sort (char* array[], int size,
          bool (&rComp)(char*, char*),
          void (&rSwap) (char*[], int, int)) {
    bool sorted;
    do {
        sorted = false;
        for (int i = 0; i < size-1; i++) {
            if (rComp(array[i], array[i+1])) {
                rSwap(array, i, i+1);
                sorted = true;
            }
        }
    }
    while (sorted);
}
```

# Reference to function

```
void swap (char* array[], int pos1, int pos2) {
    char* aux = array[pos2];
    array[pos2] = array[pos1];
    array[pos1] = aux;
}

bool compare (char* s1, char* s2) {
    if (strcmp(s1, s2) > 0) return true;
    return false;
}

int main () {
    char* tab [3] = {"bleu", "rouge", "vert"};
    sort(tab, 3, compare, swap);
    print(tab, 3);
}
```

# passing **large** objects by value

```
class V {  
public:  
    int buffer [1000];  
};  
V foo (V s) {  
    .../...  
    return s;  
}  
int main () {  
    V x;  
    V y = foo (x);  
    x = y;  
}
```

a lot of buffers are allocated

## memberwise copy

The **copy** is done by **memberwise copy**

At the call of **foo**: the **entire contents** of **x** is **copied and pushed** on the **program execution stack** and named **s**

When **foo** returns : the entire contents of **s** is **popped** out of the **program execution stack** but a copy is pushed again in the stack and named **y**

**y** is copied in **x**

# what happens in this program ?

```
class IntStack {  
    int* stack;  
    int size;  
    int top;  
public:  
    IntStack (int size) {  
        size = s;  
        stack = new int [size];  
        top = 0;  
    }  
    ~IntStack () {  
        delete [] stack;  
    }  
};  
int main () {  
    IntStack intStack (1000);  
    {  
        IntStack anotherIntStack = intStack;  
    }  
    intStack.Push(10);  
}
```

# the IntStack

intStack is memberwise copied in anotherIntStack

anotherIntStack is an **automatic object** so at the **end** of its **block**:

- it is **popped** out of the **execution stack**
- the **destructor** of IntStack is called for anotherIntStack
- anotherIntStack->stack is deleted []

you have a **serious memory problem**



```
void foo (IntStack stack) {  
}  
IntStack foo () {  
    IntStack stack(10);  
    return stack;  
}  
int main () {  
    IntStack intstack1 (100);  
    IntStack intStack2 = foo (intstack1);  
}
```

Same **problem occurs** :

when **returning** by **value** from a function

# the **copy** constructor

The **problem occurs** because the **compiler decides how** to create a **new object** from an **existing** one

And the **compiler decides** to **perform** this **creation** with **memberwise copy**

**You** can **decide how** you want **copy** to be **done** by defining your **own function** to be **used**

This function is called the **copy constructor**

This **copy constructor** is **essential** to **control passing** and **returning** of **user-defined types** by **value** during **function call**

here comes the **reference**

the copy constructor has a **single argument**

the argument and the object you are **constructing** have the same kind of *type*

the argument **cannot** be **passed** by **value** :

you are currently **defining** the way your class handles the **passing** by **value** operation

**you must use reference**

The **prototype** of a **copy constructor** for a class T is :

```
T::T (const T&)
```

the **const** qualifier avoid the copy constructor to modify the object being copied

# Example of copy constructor

```
class IntStack {  
    int* stack;  
    int size;  
    int top;  
public:  
    IntStack (int s) { /* optimized out */ }  
    ~IntStack () { /* optimized out */ }  
    IntStack (const IntStack&);  
};  
IntStack::IntStack (const IntStack& intStack) {  
    size = intStack.size;  
    stack = new int [size];  
    top = intStack.top;  
    for (int i = 0; i <= top; i++) {  
        stack[i] = intStack.stack[i];  
    }  
}  
int main () {  
    IntStack myStack1 (100);  
    { IntStack myStack2 = myStack1; }  
}
```

# preventing pass-by-value

you can prevent the **pass-by-value** mechanism for a **user-defined type** by declaring (no need to define) a **private copy constructor**

```
class X {  
private:  
    X (const X&); // 3  
public:  
    X () {}  
};  
void foo (X) {}  
int main () {  
    X x1;  
    X x2 = x1; // 10  
    foo (x1); // 11  
}  
f.cxx: In function 'int main()':  
f.cxx:3: error: 'X::X(const X&)' is private  
f.cxx:10: error: within this context  
f.cxx:3: error: 'X::X(const X&)' is private  
f.cxx:11: error: within this context
```

# copy const for Date

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
public:  
    Date (int, int, int);  
    void Tomorrow ();  
    void Print (); // on standard output  
};  
int main () {  
    Date d1;  
    {  
        Date d2;  
        d1 = d2;  
    }  
}
```

ok : the **compiler** uses **memberwise copy** to **assign** Date objects

# what happens in this program ?

```
class IntStack {  
    int* stack;  
    int size, top;  
public:  
    IntStack (int size) {  
        size = s;  
        stack = new int [size];  
        top = 0;  
    }  
    ~IntStack () { delete [] stack; }  
};  
int main () {  
    IntStack intStack1 (10);  
    {  
        IntStack intStack2 (20);  
        intStack1 = intStack2;  
    }  
    intStack1.push(12);  
}
```

# the IntStack

intStack2 is **memberwise** copied in intStack1

the **dynamically allocated array** intStack1->stack is lost

intStack1 and intStack2 share now the stack pointer

the **automatic object** intStack2 is **suppressed** at the **end** of its **scope**

the **destructor** of IntStack is called for intstack2

intStack2->stack is deleted []

intStack1->stack is **corrupted**

you have a **serious memory problem**



# assignment operator

Control user-defined type **assignment** by overloading the **operator=**

```
IntStack& IntStack::operator= (const IntStack& auxStack) {  
    delete [] stack; // do not forget to destroy this  
    size = auxStack.size;  
    stack = new int [size];  
    top = auxStack.top;  
    for (int i = 0; i <= top, i++)  
        stack[i] = auxStack.stack[i];  
    return *this;  
}
```

# assignment operator

the code of `X& X::operator=(const X&)` is quite similar to the code of `X::X(const X&)`

a solution :

- implement a function **clone**
- implement a function **reset**
- call these functions in the **copy constructor**, the **assignment operator** and the **destructor**

# chain assignment operators

return `*this` to **chain assignments** like for **built-in types**

```
void foo () {  
    IntStack stack1 (100);  
    IntStack stack2 (20);  
    IntStack stack3 (30);  
    stack1 = stack2 = stack3;  
    // stack1.operator=(stack2.operator=(stack3))  
}
```

# test for **self** assignment

```
void foo () {  
    IntStack stack (100);  
    stack = stack;  
}
```

## Always check for self-assignment in operator=

```
IntStack& IntStack::operator= (const IntStack& auxStack) {  
    if (this != auxStack) {  
        .../...  
    }  
    return *this;  
}
```

# constructor , destructor and operator=

In **user-defined** type:

If you have **implemented** a **copy constructor**, you surely **need** a **destructor** and an **operator=**

If you **really don't** want **people** to **perform assignment**: **declare** (do not define) your class **operator=** as **private**

## 16. TP2 : Continuing the implementation of your integers stack (1)

- add a copy constructor

## 17. Constructor Initializer List



```
class IntStack {  
    int* stack;  
    int size;  
    int top;  
public:  
    IntStack (int);  
    void Push (int);  
};  
class X {  
    IntStack mystack;  
public:  
    X (int i) {  
        mystack.Push(1);  
    }  
}
```

What happens in this program ?

There is no **matching function** for call to **initialize** the mystack member of X objects

The two existing functions are :

IntStack::IntStack(const IntStack &)

IntStack::IntStack(int)

# the constructor **initializer list**

C++ introduce the notion of **constructor initializer list**

It is a list of "**constructor calls**" and **initializations**

This **list** occurs **before** the **constructor body** and **after** the constructor **arguments list** (you may need arguments values)

When you **enter** the **constructor body**, **initializations must** have been **performed** so that you can **use** the **corresponding data** in a **safe** way

# initializer list for **built-in** types

Use this **way** of **initializing built-in** types

It is a **simple assignment**

It is a **good coding style**

```
class Integer {  
    int integer;  
public:  
    Integer (int i = 0) : integer(i) {}  
};
```

## initializer list for **user-defined** types

```
class X {  
    int size;  
    IntStack mystack;  
public:  
    X (int);  
};  
X::X (int s) : mystack (s), size (s) {}
```

In the **initializer list** of `X::X`, you **specify** the **arguments** that the **compiler** must **pass** to the **constructor** of `IntStack` to create the **object** `mystack`

This **constructor** is called **before** the **body** of the **constructor**

**C++** guarantee that way **proper initialization** during **composition** (and **inheritance**: it will be the manner to pass arguments to base class constructors)

```
class X {  
public:  
    int size;  
public:  
    IntStack v1;  
    IntStack v2;  
    IntStack v3;  
    IntStack v4;  
    X (int s) : v4(), v2 (v1.Size()), v1 (s+10), v3 (s) { size = s; }  
};  
int main () {  
    X x(2);  
    cout << x.v1.Size() << endl;  
    cout << x.v2.Size() << endl;  
    cout << x.v3.Size() << endl;  
}
```

If an **object** has a **default constructor**, the compiler **call** it **implicitly**, but **you** can **call** it **explicitly**

# order of **constructor** and **destructor** calls in composition

The **initializer list** can be presented in **any order**

**Constructors** are called in the **order** they are **specified** in the **class declaration** (**v1, v2, v3, v4** and **X** but **not v4, v2, v1, v3** and **X**)

The **order** of **destructor** calls for **member object** is also **unaffected** by the **order** in the **constructor initializer list**

The **order** of the **destructors** is also **determined** by the **order** of the **declarations** in **reverse order X, v4, v3, v2, v1**)

# copy constructors calls in **composition**

```
class A {
public:
    A () { cout << "A::A()\n"; }
    A (const A&) { cout << "A::A(constA&)\n"; }
};
class B {
public:
    B () { cout << "B::B()\n"; }
    B (const B&) { cout << "B::B(constB&)\n"; }
};
class X {
    A a; B b;
public:
    X () { cout << "X::X()\n"; }
    X (const X& x) : b(x.b), a(x.a) { cout << "X::X(constX&)\n"; };
int main () {
    X x;
    X y = x; }
// output: A::A() B::B() X::X()
// output: A::A(constA&) B::B(constB&) X::X(constX&)
```

# use initializer list

```
class X {  
    String str;  
    Date date;  
public:  
    X (String s, Date today) : date(today) {  
        str = s;  
    }  
};
```

**date** is **initialized** with a **copy** of today

**str** is **first initialized** with the **default constructor** (no need) and then assigned



## 18. Static

# static

In C and in C++ **static** has two meanings

- **static storage:** allocation at a fixed address in the static data area (global objects are static)
- **local visibility:** the name is local to a particular translation unit or to a particular structure (a static function)

In C and C++ if you do **not provide** an **initializer** for a **static variable** of a **built-in type**, the compiler **guarantee** that variable will be initialized to 0 (converted to the proper type)

**static** object of **user-defined** types are initialized with constructor calls

If you do **not specify constructor** arguments, the class must have a **default constructor**

# static variable in functions

When you create a variable inside a function

- the compiler allocates storage on the execution stack
- the compiler performs initialization if you ask him to

If you want to **retain a value between function calls** and if you want that value to be **under the scope of the function**:

- you create a **static variable inside the function**
- the storage is in the **static data area**
- the object is **local** to the **function**
- the object is **initialized the first time the function is called**
- the **variable retains** its **value** between **function invocations**

# the number of function call occurrences

```
void foo () {  
    static Date today;  
    static IntStack stack (120);  
    static int numberOfFooCalls;  
    numberOfFooCalls++;  
}
```

## Initializations:

- **Date** must have a default constructor
- **IntStack** a constructor with one **int** argument
- **numberOfFooCalls** will be by default initialized to **0**

The initialization is done the first time **foo** is called but the **storage** is done **before** the **main** is entered

# static object **destruction**

**Destructors** for **static** objects are called:

- when **main** exits
- when the function **exit** is explicitly called

But **destructors** are not called when program is exited with **abort**

**Destruction** of static objects occurs in the **reverse order** of their **initialization**

Of course if a **function** containing a **static local object** is **never called**, the **constructor** is **never executed** (the storage is done) so the **destructor** is **not executed**

# static (statically allocated) class member

Each object of a **structure** (struct or class), has its **own copy** of **each non static data member** of the structure

If you want **all objects of a structure** to share a **single storage space**:

- Do not use **global variable**, it is not safe, anyone can modified a global variable
- Use **static data members**
- All **objects** of the **structure** share the **same static storage space** for that data member
- The **static member** is **scoped** inside the **class** and it can be **public**, **private** or **protected**

```
class Vertex {  
    static int radius;  
    // .../...  
};  
// in the class implementation file  
int Vertex::radius = 12;
```

In the class **Vertex** the static data member **radius** is **declared** but not **defined**

You must **define** it somewhere **outside** the **class** in an implementation file

Of course, only **one definition** is allowed

```
class X {  
    static const int size;  
    static const int table [4];  
    static char buffer [3];  
};  
// in the class implementation file  
const int X::size = 5;  
const int X::table[4] = {0, 1, 2, 3};  
char X::buffer [3] = {'a', 'b', 'c'};
```

Linkage refers to elements having an address at compile time



# static constants inside classes

```
class X {  
    static const int size;  
    int array [size];  
};  
// in the class implementation file  
const int X::size = 100;
```

It is the way in C++, to define **compile-time constants**

# static member functions

You can create **static member functions**

A **static** member function **cannot access ordinary** data members, **only static** data members

**static** member functions can only **call** other **static member functions**

**static member functions** do not have **this**

**static members** (data or function) are **associated** with a **class** and **not** with a particular **object**

```
class Vertex {  
private:  
    static int radius;  
public:  
    static int getRadius () { return radius; }  
    static void setRadius (int r) { radius = r; }  
    // .../...  
};  
int Vertex::radius = 12;  
int main () {  
    int i = Vertex::getRadius();  
    Vertex v;  
    int j = v.setRadius(10);  
}
```

To **call static member** functions and data, you **don't need** that an **object** of this class **exists**

The use of **static members** reduces the **need** for **global elements**

Use it to **package global variables** and **functions** in classes

## **important** use of statically allocated classes objects

The **constructor** for a **global static object** is called **before** the **main** is entered

This the way in C++ to **execute code before entering** the **main** and **after exiting** it

There is an important use of static global object in C++:

You are sure that **important objects** like the **standard input/output streams** (**cin**, **cout**, **cerr**) are **created** and **initialized before** any possible **use** of them

## 19. Const

# Restriction in the use of **const**

```
// file.cpp
#include <string.h>
// We include the c standard library for string manipulation
// because we need the function of string comparison
// Suppose the prototype found is: strcmp (char* s1, char* s2);

void foo (const char* s1, const char* s2);
    if (strcmp (s1, s2)) { // compile time error
        // You try to break the constness of s1 and s2.
        .../...
    }
```

# Const Object inside Class Declaration

The **compiler** allocates **storage** for **const** object, in **each class object**

This **value must** be **initialized once** and then **never changed**

It **becomes** a **constant** for the **lifetime** of the **object**

When you **enter** the **body** of the **constructor**, the **const member data** must have been **initialized** otherwise it is a **compile-time error**

# Initialization of Const Member Object

```
class IntStack {  
    const int size;  
    int* stack;  
public:  
    IntStack (int);  
};  
IntStack::IntStack (int s) : size(s) {  
    stack = new int [size];  
}
```

The **initializer list** is important for **built-in** types too



# Constructor Initialization List for Reference Data Members

The same **problem** occurs when **data member** are **reference**

A **reference** must be **initialized** before **use**, the initializer list is the **only place** to **initialize** it

But it is **not constructor call**

```
IntStack GlobalIntStack (12);  
class X {  
    IntStack& stack;  
public:  
    X () : stack (GlobalIntStack) {}  
};
```

# Compile Time Constant in Class

**Const member does not allow compile-time constant in class: use static member**

```
class IntStack {  
    const int size = 100;           // error  
    int stack [size];  
};  
  
class IntStack {  
    static const int size = 100;    // ok  
    int stack [size];  
};
```

# Using Member Functions for Const Objects

```
class Date {  
private:  
    int day, month, year;  
public:  
    int i;  
    Date (int, int, int = 1998);  
    void SetDay (int i) { day = i; }  
    void Next();  
    void Print();  
};  
int foo () {  
    const Date birthday (2, 4);  
    birthday.i = 12;    // assignment of read-only member 'Date::i'  
    birthday.Print();  // 'const Date' as 'this' argument of  
                        // 'void Date::Print()'  
    birthday.SetDay(12); // 'const Date' as 'this' argument of  
                        // 'void Date::SetDay(int)'  
}
```

# Using Member Functions for Const Objects

The **compiler** checks that **no data member** of **const** objects **changes** during the **object lifetime**

The **compiler** checks easily that no **public** member is modified through const objects

But it **does** not **know which member function changes** the data and **which** one are **safe** for a **const** object

# Const Member Functions

You tell the **compiler** that a **member function** does **not modify this** by **declaring** this function **const**

```
class Date {  
private:  
    int day, month, year;  
public:  
    int i;  
    Date (int, int, int = 1998);  
    void Print() const;  
};
```

```
void Date::Print () const { /*...*/ }  
int foo () {  
    const Date birthday (2, 4);  
    birthday.Print();  
    Date date (2, 4);  
    date.Print();  
}
```

# Const Member Functions

A **const member function** can be **called** for **const objects** and for **variable objects**

An **ordinary member function** can **only** be **called** for **variable objects**

A **const member function** is **declared** to be able to **read** but **not write** the object (**this**) for which it is called

```
class Date {  
private:  
    int day, month, year;  
public:  
    Date (int, int, int = 1998);  
    void Print() const;  
};  
void Date::Print () const {  
    day ++; // In method 'void Date::Print() const':  
           // increment of read-only member 'Date::day'  
}
```

The **type** of **this** in a **const** member **function** of class **X** is :  
**const X \* const this**

It is a **logical const** (“appears constant to its user”) not a **physical** (“stores in read only memory”)

**Any function that does not modify data members should be declared const** : they can be used for const objects

# Changing data member of const object

When you **want** to **change** some particular **data** of **this** inside a **const function**

You can **cast away constness**, by **casting this** to a **pointer** to a **variable object**

```
// in a Date.h file
class Date {
    int day, month, year;
public:
    Date (int, int, int = 1998);
    void Print() const;
};

// in a Date.cpp file
void Date::Print () const {
    day++;
    // compile time error
    ((Date*) this)->day++;} // ok the cast removes constness
```



# Casting away constness

A **user** of **your class Date**, may **not** have **access** to the **code** of **Print** and he has **no idea** that you **modify this** in a **const function**

Do **not remove constness** with **cast** (`const_cast`)

# mutable keyword

You can **specify** that a **particular data member** class can be **modified inside const member functions** by **declaring it mutable**

```
class X {  
    int i;  
    mutable int j;  
public:  
    void f () const;  
};  
void X::f () const {  
    i++; // increment of read-only member 'X::i'  
    j++; // ok  
}
```

Now the **user** of a **class** can see from the **class declarations part** which **data members** can be **modified** in **const member functions** mutable specifier can be applied only to names of class data members

- mutable cannot be applied to names declared const or static
- mutable cannot be applied to reference members

## 20. Inline Functions

# Problem of Efficiently

The **macros** in C, allows to make a **piece of code** looks like a **function call without** the **overhead** of a **real function call**

You **avoid** the cost of the function call : **pushing** arguments, **returning** argument, ...

In the **macros** all the **work** is done by the **preprocessor**: it **replace** the macro call **directly** with the **macro code**

The **main** problem with **macros** in C++ is that a **macro** is not **type-checked**

# inline functions

The idea, in C++, was to bring a kind of macros under the control of the compiler

In C++ you have **inline** functions

An **inline** function **behaves** like an **ordinary function** but is **expanded in place** during **compilation**

The **overhead** of the **function call** is **eliminated**

```
inline bool min (int a, int b) { return a < b; }  
inline bool min (char* a, char* b) {  
    return strcmp(a, b) < 0 ? true : false;  
}
```

In C++: **avoid** the **use of macros** , prefert **inline functions**

# Inline Member Functions

A **member function** defined in the **class body** is **automatically inline**

```
class X {  
    Date date;  
public:  
    Date GetDate () const { return date; }  
    void SetDate (Date newDate) { date = newDate} ;  
};
```

The **most important use** of **inline member functions** is to **implement data member accessors** (read, write of field)

A **data member** can be made **private** and its access **controled** by the **class** through **accessors without overhead** of **function call** with the use of **inline**

# Inline Member Functions

When the **compiler cannot inline** a function it simply **returns** to the **ordinary function call** (the function is too complicated, you use the address of the function somewhere, ...)

An **inline** is a **suggestion** for the **compiler** but the **compiler** is **not forced** to **inline** every thing you **ask** it to

```
// in file IntStack.h
class IntStack {
    int* stack;
    int size;
    int top;
public:
    IntStack (int);
    ~IntStack ();
    void Push (int);
    int Top ();
    int Pop ();
    bool Empty ();
    bool Full ();
};
```

In **class**, most of the time, a **member function** is **provided** where a **C program** would **simply** contain the **small** piece of **corresponding code**

The **cost** of **calling** the **member functions** is **too high** compared to the **cost** of simply executing the **body**: **do not forget inline**



```
// in file IntStack.h (continued)
inline IntStack::IntStack (int s) {
    top = 0;
    size = s;
    stack = new int [size];
}
inline IntStack::~IntStack (int s) {
    delete [] stack;
}
inline void IntStack::Push (int i) {
    stack[top++] = i;
}
inline int IntStack::Top () {
    return stack[top+1];
}
inline int IntStack::Pop () {
    return stack[--top];
}
inline bool IntStack::Full () { return top == size; }
inline bool IntStack::Empty () { return top == 0; }
```

## 21. TP2 : Continuing the implementation of your integers stack (3)

- add a const size
- use initializer list
- add constness to functions where you can
- add an operator=
- push, pop, empty, full, constructor must become inline functions
- implement the operator <<

## 22. Exception Handling

# Techniques to handle a problem

The author of a library can detect a run-time error (zero divide)

When detecting a problem, his program could :

- 1 terminate  $\Rightarrow$  *the library cannot be used in an embedded program that cannot afford to crash*
- 2 return a value representing the error  $\Rightarrow$  *you cannot avoid this value to be taken as a plausible result*
- 3 return a legal value and leave the program in an illegal state  $\Rightarrow$  *the calling function may not notice that the program is now in a illegal state*
- 4 call a function supplied to be called in case of error (you know how to cope with a problem : it is not an exception)

# Fundamental idea

When you don't know what to do about a problem : give the control to the caller by using the notion of exception (the user of a library surely knows how to cope with errors)

A function that finds a problem that it cannot cope with **throws** an exception

A function that wants to handle that kind of problem can indicate what to do by **catching** that exception

the exception mechanism is an alternative to traditional techniques

# Throwing an Exception

```
class SizeError {};  
class Vector {  
    int* v; int size;  
public:  
    class RangeError {};  
    Vector (int s) {  
        if (s < 0) {  
            throw SizeError ();  
        } else {  
            v = new int [s];  
            size = n;  
        }  
    }  
    int& operator[] (int index) {  
        if (0 <= index && index < size) return v[index];  
        throw RangeError ();  
    }  
};
```

**code that detects** an error **throws** an **object**

# Catching an Exception

Enclose the code in which you want to **catch** an error(range, size, ...) in a **try** statement with an exception handler **catch**

```
void foo (Vector& v) {  
    try { Vector v1 (-12);          // if your code causes an exception  
        v[100] = 13;                // the exception will be caught by the handler  
    }  
    catch (Vector::RangeError) {  
        fixit (v) ; // adjust the use and for example try again  
    }  
    catch (SizeError) {  
        exit (99);  
    }  
    // We get here if no SizeError exception appear or if a  
    // Vector::RangeError exception was handled  
}
```



# The Exception Handler **catch**

It can be used only immediatly after a **try** keyword or after another **catch**

The **()** contain a declaration that is used like a function argument

If a function throws an exception for which there is no matching exception handler the program will call the function **void std::terminate ()**

# Naming Exceptions

An exception is caught by specifying its **type**

But what is throw is not a type but an **object**

If you need to transmit extra informations from the **throw** point to the **handler** point, **put some data into the object**

The construct after the **catch** specifies what exception type is acceptable and optionally names the exception

```
class Vector {
    friend ostream& operator<< (ostream&, Vector&);
    int size;
    int* v;
public:
    struct RangeError {
        Vector& vector;
        int index;
        RangeError (Vector& v, int i) : vector (v), index(i) {}
    };
    int& operator[] (int s) {
        if (0 <= s && s < size) return v[s];
        throw RangeError (*this, s);
    }
};

ostream& operator <<(ostream& os, Vector& v) {
    os << "Vector [" << v.size << "];"
    return os;
}

void foo (Vector& v) {
    try { v[100]; }
    catch (Vector::RangeError r) {
        cerr << r.vector << "bad index " << r.index << endl;
    }
}
```

# Exceptions Member of more than One Group

```
class Network_File_Error : public Network_Error,
                           public File_System_Error {
    ...
};

void foo () {
    try { /*...*/ }
    catch (Network_File_Error) { /*...*/ }
}

void bar () {
    try { /*...*/ }
    catch (File_System_Error) { /*...*/ }
}
```

# Derived Exception : Use Virtual if Needed

```
class MathError {  
    virtual void DebugPrint ();  
};  
  
class Overflow: public MathError {  
    virtual void Debugprint ();  
};  
  
void foo () {  
    try { /*...*/ }  
    catch (MathError& m) {  
        m.Debugprint();  
    }  
}
```

# Re-Throwing Exception

```
void foo () {  
    try { ... }  
    catch (MathError) {  
        if (CanHandleIt) {  
            // handle  
        } else {  
            throw; // re-throw the exception caught  
        }  
    }  
}
```

A re-throw is indicated by a **throw** without an argument : the exception re-throw is the original exception caught and not just the part of it that was accessible as a **MathError**

## To **catch** any exception : Use **ellipsis**

```
void foo () {  
    try { /*...*/ }  
    catch (...) {  
        // cleanup  
        throw;  
    }  
}
```

Because a derived exception can be caught by many handlers the order in which the handler are written in a **try** statement is significant : **the handlers are tried in order**

# Resource Acquisition

When a function requires a resource, for example opens a file, it is important that the resource is properly released

```
void UseFile (const char* filename) {  
    FILE* file = fopen(filename, "w");  
    /*...*/    // use file  
    fclose (file);  
}
```

If something goes wrong after the **fopen** and before the **fclose** an exception may cause the function `Usefile` to be exited without calling `fclose` because of the **jump** of **throw**



# Use constructor and Destructor to Handle Acquisition and Release

When you use **file**, even if an exception is **thrown**, **file** will be closed by the automatic call - file is an object - to the destructor **FilePtr()** when the program exit the function

```
class FilePtr {
    FILE* file;
public:
    FilePtr (const char* filename, const char* mode) {
        file = fopen (filename, mode);
    }
    FilePtr (FILE* f) { file = f; }
    ~FilePtr () { fclose (file); }
    operator FILE* () { return file; }
};

void UseFile (const char* filename) {
    FilePtr file (filename, 'w');
    ...
}
```

# Interface Specifications

Specify a set of exceptions that might be thrown as part of function declaration

```
void foo () throw (x1, x2, x3); // function declaration
```

**foo()** may throw exceptions **x1**, **x2**, **x3** and exceptions derived from these types but not others

If during execution **foo()** throws another exception, a call to **std::unexpected()** will be done

By default **std::unexpected()** calls **std::terminate()**, which in turn normally calls **abort()**

# Program Termination

a program can terminate :

- by returning from `main`
- by throwing an **uncaught** exception
- by calling `exit()` :
  - static object destructors are called so calling `exit` in a destructor may cause an infinite recursion
  - destructors of the "calling function local variables" are not invoked (best to use exception)
- by calling `abort()` (static object destructors are not called)

# std::set\_unexpected and std::set\_terminate

**std::set\_unexpected** change the behavior of **std::unexpected()**

It takes the address of a function with no argument and void return value

It returns the previous value of the **unexpected** function pointer so you can save it and restore it

You can also install your own **terminate()** function using **std::set\_terminate** which returns the pointer to the **terminate()** function you are replacing

```
#include <iostream>
void (*old_terminate)() = set_terminate(my_terminate);
```

# Explicit Declaration of Exceptions in Function

```
void foo () throw (x1, x2, x3) { /* some code */ }  
// is equivalent to writing this :  
void foo () {  
    try { /* stuff */ }  
    catch (x1) { throw; }  
    catch (x2) { throw; }  
    catch (x3) { throw; }  
    catch (...) { unexpected (); } // by default calls terminated  
}
```

```
void foo () throw (); // will throw no exception
```

A function with no exception-specification allows all exceptions

A function with an empty exception specification, throw(), does not allow any exception

# Avoiding `abort()`

to ensure cleanup of uncaught exception, add a catch-all handler to `main()`

```
int main () {  
    try {  
        /*...*/  
    }  
    catch (my_exception) {  
        /*...*/  
    }  
    catch (...) {  
        /*...*/  
    }  
}
```

it does not **catch exceptions** thrown **by destructors** of **global static** variables

# Exception in Constructor

When a constructor throws an exception, the dynamic memory is not allocated

```
class X { /*...*/ };  
int main () {  
    X* x1 = new X;  
    X* p2 = new X[10];  
}
```

# Exception in Initializer List (pb)

When a **member initializer** throws an exception :

[1] by **default** the **exception** is **passed** to the **code** invoking the constructor

[2] the constructor can catch the exception



# Exception in Initializer List (pb)

```
class X {  
    Vector v;  
public:  
    X (int s)  
    try  
        : v(s)  
    {  
        // constructor code  
    }  
    catch (Vector::size) {  
        // catch it  
        // but it is automatically rethrown  
    }  
};
```

# Exception in Destructor

When a **destuctor** throws an exception :

- 1 there is no problem if it was the **normal call** of a destructor (automatic objects exiting from a scope, call of delete...)
- 2 if the destructor was call during exception handling (the exception-handling mechanism calls somewhere a destructor) then it is a failure of the exception-handling mechanism and `std::terminate` is called

# Exception in Destructor

protect the destructor against throwing exception

```
X::~X () {  
    try {  
        /*...*/  
    }  
    catch (...) {  
        /*...*/  
    }  
}
```

## 23. TP2 : Continuing the implementation of your integers stack (4)

- add exceptions when you pop an empty stack or push a full one or give a negative size, ...

## 24. Namespace

# namespace definition

a namespace is an **optionally-named** declarative region

```
namespace my_namespace {  
    class List {};  
    int Nb = 100;  
    void foo (int n) {}  
}  
int main () {  
    my_namespace::List l;  
    my_namespace::foo (my_namespace::Nb);  
}
```

# Using Declarations

When a **name** is **frequently** used **outside** its **namespace**, it can be **imported** in **scope** with a **using** declaration

```
namespace my_namespace {  
    class List {};  
    int Nb = 100;  
    void foo (int n) {}  
}  
  
int main () {  
    using my_namespace::List;  
    List l;  
    my_namespace::foo (my_namespace::Nb);  
}
```



# Using Directive

When you want all the names of a namespace to be known in some other place use the **using namespace** directive

```
namespace MyNameSpace {  
    class MyClass {};  
    void myFunction () {}  
}  
  
using namespace MyNameSpace;  
  
void foo () {  
    MyClass* m = new MyClass();  
    myFunction();  
}
```

# Unnamed Namespace

When you don't want a namespace name to be known outside a local context simply leave the namespace without name

```
namespace {  
    class MyClass {};  
    void myfunction () {}  
    void myfunction (MyClass*) {}  
}
```

```
namespace XXX {  
    class MyClass {};  
    void myfunction () {}  
    void myfunction (MyClass*) {}  
}  
using namespace XXX;
```

The two are equivalent but there is no way of naming a member of an unnamed namespace from another translation unit only inside the whole file defining it.

## global scope use ::

**Global variables** and **global functions** are **not hidden** by **member variables** or **member functions** : you can **access them** by **qualifying it** with the **global scope** operator

The **compiler** searches first "locally" before looking for a more **global version** of this **name**

```
int i;  
struct X {  
    int i;  
    void set ();  
};  
void X::set () { i = ::i; }
```

the **definition** of a namespace can be **split** over **several** parts of one or more translation unit

```
namespace my_namespace {  
    class List {};  
    int Nb = 100;  
    void foo (int n) {}  
}  
void bar () {  
    my_namespace::List l;  
    my_namespace::foo (my_namespace::Nb);  
}  
namespace my_namespace {  
    class MyClass {};  
    void myFunction () {}  
    MyClass m;  
}
```

# Namespace Aliases

```
namespace AVeryLongNameForANamespace {  
    int NUM = 12;  
    void foo () {}  
    void bar () {}  
}  
namespace bob = AVeryLongNameForANamespace;  
int main () {  
    bob::NUM = 123;  
}
```

# Choose Between Namespace and Class

A namespace is a simple named scope

A class is a type defined by a named scope that describes how an objects of that type can be created and used

Namespaces are preferred over classes when :

- all that is needed is encapsulation of names
- type checking and object creation is not needed
- a namespace is open and a class is closed

## 25. TP2 : Continuing the implementation of your integers stack (5)

- add a namespace



## 26. Operator Overloading

# Introduction

The C++ data type **int** together with **+**, **-**, **\*** and **/** **provides a restricted implementation** of the mathematical **integer type**

in C++ classes provide facilities for specifying nonprimitive objects representation together with a set of operations that can be performed on such objects

**In C++** you can **implement** these **operations** by **overloading C++ native operators**

**Operator overloading** is just “**syntactic sugar**”: it is simply **another way** for you to **make function call**

**The difference** is in the **function call syntax**: it is the **same** as **operator call** on **built-in types**

# Operator functions

In a **class**, you can **redefine** the **meaning** of the following **operators**

```
new delete new[] delete []
+ - * / % ^ & | ~ !
= < > += -= *= /= %= ^= &=
|= << >> >>= <<= == != <= >= &&
|| ++ -- ->* , -> [] ()
```

Your **cannot** redefined the following **operators**

```
:: .* .
?: # ##
```

```
sizeof typeid dynamic_cast
const_cast reinterpret_cast
```

# Operator overloading

The **Predefined meaning** of **built-in types operator** cannot be **redefined** (you cannot redefine `+` for `int`): operator overloading is only possible on user-defined types

- You **cannot change** the **precedence**: you must use parentheses
- You **cannot** change the **operator arity**: **!** **cannot** be made binary
- **Operator functions cannot** have **default arguments**
- You **cannot** define a **novel operator** ( `**` for power)

# Operators : Member or Global Functions

You can **define** it as a **member function**

you can **define** it as a **global function taking argument**

The **operator syntax** and the **function call notation** are **different**:

expression	as member function	as non-member function
<code>++a</code>	<code>(a).operator++ ()</code>	<code>operator++(a)</code>
<code>a+b</code>	<code>(a).operator+ (b)</code>	<code>operator+(a, b)</code>
<code>a++</code>	<code>(a).operator++ (0)</code>	<code>operator++(a, 0)</code>

# Binary Operator **a + b** Definition

A member function taking one argument (the second is **this** )

```
class X {  
    X operator + (X);  
};
```

**a + b** is then interpreted by **a.operator + (b)**

A global function taking two arguments

```
X operator + (X, X);
```

**a + b** is then interpreted by **operator + (a, b)**

# Prefix and Postfix Unary Operators Definition

A member function taking no argument (the **argument** is **this** )

```
class X {  
    X operator ++ ();      // prefix  
    X operator ++ (int);  // postfix  
};
```

**++a** is interpreted as **a.operator ++ ()**

**a++** is interpreted as **a.operator ++ (0)**

A global function taking one argument

```
X operator ++ (X);        // prefix  
X operator ++ (X, int);  // postfix
```

**++a** is interpreted as **operator ++ (a)**

**a++** is interpreted as **operator ++ (a, 0)**

# Arguments and this : Const or Non-Const ? You see...

if you need to **read** and **not** to **write** arguments : pass them as **const**

if you do not modify this implement const **member functions** or **global functions**

to **access private data members** use **friend** (**friend** declarations are very useful for operator overloading)



# Output Operator

```
#include <iostream>
using namespace std;

class Integer {
    friend ostream& operator<< (ostream& os, const Integer&);
    int integer;
public:
    Integer (int i) : integer(i) {}
};

ostream& operator<< (ostream& os, const Integer& i) {
    os << i.integer;
    return os;
}

int main () {
    Integer i(12), j(13);
    cout << i << " " << j << endl;
}
```

# Comparison Operators

they do not change their **arguments**

they do not modify **this**

```
class X {  
public:  
    bool operator< (const X& x) const {...}  
    bool operator== (const X& x) const {...}  
    bool operator!= (const X& x) const {...}  
};
```

```
bool operator< (const X& x1, const X& x2) {...}  
bool operator== (const X& x1, const X& x2) {...}  
bool operator!= (const X& x1, const X& x2) {...}
```

# Member Auto Assignment

they do not change their **right argument** : pass **right** argument as **const**

they change their **left value** : implement **non-const member functions**

```
class Integer {  
    int integer;  
public:  
    Integer (int i) : integer(i) {}  
    Integer& operator+= (const Integer& i) {  
        integer+=i.integer;  
        return *this;  
    }  
};
```

```
int main () {  
    Integer i(12), j(13);  
    i+=j;  
}
```

# Global Auto Assignment

they do not change their **right argument** : pass **right** argument as **const**

they change their **left value** : implement **global functions**

```
class Integer {  
    friend Integer& operator+= (Integer&, const Integer&);  
    int integer;  
public:  
    Integer (int i) : integer(i) {}  
};  
inline Integer& operator+= (Integer& i1, const Integer& i2) {  
    i1.integer+=i2.integer;  
    return i1;  
}
```

```
int main () {  
    Integer i(12), j(13);  
    i+=j;  
}
```

# Member Prefix Auto Decrement

```
class Integer {  
    int integer;  
public:  
    Integer (int i = 0) : integer (i) {}  
    Integer& operator-- () {  
        integer = integer-1;  
        return *this;  
    }  
};  
int main () {  
    Integer i = 12;  
    cout << --i << endl; // 11  
}
```

# Global Prefix Auto Decrement

```
class Integer {  
    friend Integer& operator-- (Integer&);  
    int integer;  
public:  
    Integer (int i = 0) : integer (i) {}  
};  
inline Integer& operator-- (Integer& i) {  
    i.integer = i.integer-1;  
    return i;  
}  
int main () {  
    Integer i = 12;  
    cout << --i << endl; // 11  
}
```

# Global Postfix Auto Decrement

```
class Integer {  
    friend Integer operator-- (Integer&, int);  
    int integer;  
public:  
    Integer (int i = 0) : integer (i) {}  
};  
inline Integer operator-- (Integer& i, int) {  
    Integer aux = i;  
    i.integer = i.integer -1;  
    return aux;  
}  
int main () {  
    Integer i = 12;  
    cout << i-- << endl; // 11  
}
```

# Member Postfix Auto Decrement

```
class Integer {  
    int integer;  
public:  
    Integer (int i = 0) : integer (i) {}  
    Integer operator-- (int) {  
        Integer aux = *this;  
        integer = integer - 1;  
        return aux;  
    }  
};  
int main () {  
    Integer i = 12;  
    cout << i-- << endl; // 11  
}
```



# Return Values: Const or Non-Const ? You see ...

The **constness** of the **return value depends** of the **expected meaning** of **operator**:

If your **operator produces** a **new value** (like +)

If you want to avoid modifying a temporary object

⇒ you must **return** the **new object** as **const**

# class Integer

```
#include <iostream>
using namespace std;
class Integer {
    int integer;
public:
    Integer (int i = 0) : integer (i) {}
    friend const Integer operator+ (const Integer&, const Integer&);
    Integer operator++ () {
        ++integer;
        return *this;
    }
};
inline const Integer operator+ (const Integer& arg1, const Integer& arg2) {
    return Integer (arg1.integer+arg2.integer); }
int main () {
    Integer i = 12, j = 10;
    ++(i + j);
}
i.cxx: In function 'int main()':
i.cxx:25: warning: passing 'const Integer' as 'this'
argument of 'const class Integer Integer::operator ++()'
discards const
```

# operator+

```
#include <iostream>
using namespace std;
class Integer {
    int integer;
public:
    Integer (int i = 0) : integer (i) {}
    friend Integer operator+ (const Integer&, const Integer&);
    Integer operator++ () {
        ++integer;
        return *this;
    }
};
inline Integer operator+ (const Integer& arg1, const Integer& arg2) {
    return Integer (arg1.integer+arg2.integer);
}
int main () {
    Integer i = 12, j = 10;
    ++(i + j);
}
```

# Return Values: Reference or Object ? You see ...

If your **operator returns** an object  
when chaining the operations  
you might modify a temporary object without being warning by the  
compiler !

If your **operator returns** a reference to this, or to its left argument  
when chaining the operations  
you modify an existing object

# Return Value by Reference

```
class Integer {  
    int integer;  
public:  
    Integer (int i = 0) : integer (i) {}  
    Integer& operator+= (const Integer& ri) {  
        integer+=ri.integer;  
        return *this;  
    }  
};  
int main () {  
    Integer i = 12, j = 13, k = 14;  
    (i+=j)+=k;  
    cout << i; // 39  
}
```

# Return Value by "Object"

```
class Integer {  
    int integer;  
public:  
    Integer (int i = 0) : integer (i) {}  
    Integer operator+= (const Integer& ri) {  
        integer+=ri.integer;  
        return *this;  
    }  
};  
int main () {  
    Integer i = 12, j = 13, k = 14;  
    (i+=j)+=k; // (i.operator+=(j)).operator+=(k)  
    cout << i; // 25 - Is it what you were expecting ?  
}
```

# Returning Non-Temporary Objects

For **operations** to be **chained** : **return a reference**

The **argument** is **not modified** : **pass it as const**

```
class Integer {  
    Integer& operator+= (const Integer& arg) {  
        integer+=arg.integer;  
        return *this;  
    }  
};  
  
int main () {  
    Integer i = 12;  
    Integer j = 10;  
    Integer k = 7;  
    i+=j+=k;      // i.operator+= (j.operator+= (k))  
}
```

# Returning Non-Temporary Objects

If you **decide** that the **return value** can be **modified** : **use non-const**

```
class Integer {  
    Integer& operator+= (const Integer& arg) {  
        integer+=arg.integer;  
        return *this;  
    }  
};  
  
int main () {  
    Integer i = 12;  
    Integer j = 10;  
    Integer k = 7;  
    (i+=j+=k)++;    // ok  
}
```



# Increment and Decrement Operators

**They modify the object:** the **left value cannot be const**

The **prefix** (resp. **postfix** ) **version returns the value** of the **object after** (resp. **before** ) it **was changed**

**Should they return non-const value ?**

**No problem** for **postfix** because **postfix returns \*this**

**Problem** for **prefix** because **prefix returns a temporary object**

**For more sense** you can **make them both const**

**avoid** using **postfix increment** and **decrement** operators to **avoid temporary objects**

# operator++ return value

## Should operators++ return non-const value ?

```
class Integer {  
    int integer;  
public:  
    Integer (int i = 0) : integer (i) {}  
    Integer& operator++ () {  
        ++integer;  
        return *this;  
    }  
    Integer operator++ (int) {  
        Integer aux (integer);  
        ++integer;  
        return aux;  
    }  
};  
int main () {  
    Integer i = 12;  
    i++;  
    ++(i++); // ok but you are modifying a temporary object  
}
```

# operator++

## Should operators++ return const value ?

```
class Integer {
    int integer;
public:
    Integer (int i = 0) : integer (i) {}
    const Integer& operator++ () { ++integer; return *this; }
    const Integer operator++ (int) {
        Integer aux (integer);
        ++integer;
        return aux;
    }
};

int main () {
    Integer i = 12;
    cout << i++;
    ++(i++); // it fails
}

In function 'int main()':
error: passing
'const Integer' as 'this' argument of
'const Integer& Integer::operator++()'

```

# Efficiency in return ?

```
const Integer operator+ (const Integer& arg1, const Integer& arg2) {  
    Integer aux (arg1.integer+arg2.integer);  
    return aux;  
}
```

The **aux object** is **created** on the stack and **initialized** by a **call** to the **constructor**

The **copy constructor** copies the object into the **outside location**

The **destructor** is **called** for the **aux object**

# Efficiency in return ?

```
const Integer operator+ (const Integer& arg1, const Integer& arg2) {  
    return Integer (arg1.integer+arg2.integer);  
}
```

The **compiler knows** (may be) that your **only need** is to **return** the **created object**

It **creates** the **object directly into** the **location** of the **outside return value** with a call to the **constructor**

**No copy constructor call, no destructor call**

# class Integer

```
class Integer {  
    friend const Integer operator+ (const Integer&, const Integer&);  
    friend const Integer operator- (const Integer&, const Integer&);  
};  
const Integer operator+ (const Integer& arg1, const Integer& arg2) {  
    return Integer (arg1.integer+arg2.integer);  
}  
const Integer operator- (const Integer& arg1, const Integer& arg2) {  
    Integer aux (arg1.integer-arg2.integer);  
    return aux;  
}  
int main () {  
    Integer i = 12;  
    Integer j = 10;  
    i + j;    // constructor  
    i - j;    // constructor, copy constructor, destructor  
}
```

# Use inline !

```
class Integer {  
    friend const Integer operator+ (const Integer&, const Integer&);  
    friend const Integer operator- (const Integer&, const Integer&);  
};  
inline const Integer operator+ (const Integer& arg1, const Integer& arg2) {  
    return Integer (arg1.integer+arg2.integer);  
}  
inline const Integer operator- (const Integer& arg1, const Integer& arg2) {  
    Integer aux (arg1.integer-arg2.integer);  
    return aux;  
}
```

# Operators Predefined Meaning

Only operator `=`, `[]`, `()`, and `->` must be non **static** member functions :  
it ensures their first operand to be a **lvalue**

For **int** `++a` `a+=1` and `a = a + 1` are the same. But such relation do not hold for **user-defined operator** unless the user happens to define them that way

**operator += ()** is not deduced from the definition of :  
**operator+ ()** and **operator = ()**



# Operator and User Defined Types

An operator function must either be a **member** or take at least **one class object argument** (except for **new** and **delete**)

This ensure a user cannot change the meaning of an expression unless it contains an object of a user-defined type

C++ is **extensible** but not **mutable**

# class Integer

```
#include <iostream>
using namespace std;
class Integer {
    friend ostream& operator<< (ostream&, const Integer&);
    int integer;
public:
    Integer (int i): integer (i) {}
    friend const Integer operator+ (const Integer&, const Integer&);
    const Integer operator- (const Integer& right) const {
        return Integer (integer-right.integer); }
};
const Integer operator+ (const Integer& arg1, const Integer& arg2) {
    return Integer (arg1.integer+arg2.integer); }
int main () {
    Integer i = 12;
    Integer j = 10;
    cout << i + j << endl;
    cout << 2 + j << endl; // no problem
    cout << i - j << endl;
    cout << 2 - i << endl; // error: no match for 'operator-' in '2 - i'
}
```

# Member or Not Member ?

```
cout << 2 + j << endl;    // operator+(2, j)
cout << 2 - i << endl;    // 2.operator-(i)
```

**No class `int`** where you can **define `operator-`** to accept **`2.operator-(i)`**

**If there were** such a class, you will get **two different member functions** to **cope `2 - i`** (in the definition of `int`) and **`i - 2`** (in the definition of `Integer`)

**operator functions intended to accept a basic type as their first operand cannot be member functions : use global**

To **create an `Integer` from an `int`**, the compiler **uses the constructor `Integer::Integer (int)`**

# Type conversion

```
class X {  
public:  
    X () {}  
};  
class Y {  
public:  
    Y (X&);  
};
```

```
void foo (const Y&);  
int main () {  
    X a;  
    foo (a);} 
```

Y (X&) tells the compiler how to create an Y from an X

**constructors** taking an object of another type as its **single argument** allow the **compiler** to **perform automatic type conversion**

# explicit constructor

If you **don't** want **automatic conversion** via a **constructor** : declare it **explicit**

```
class Y {  
public:  
    explicit Y (const X&);  
};
```

```
void foo (const Y&);  
int main () {  
    X a;  
    Y y(a);    // ok  
    foo (a);   // error  
}
```

```
File.C: In function 'int main()':  
File.C:13: conversion from 'X' to non-scalar type 'Y' requested  
File.C:9: in passing argument 1 of 'foo(const Y &)'
```

# IntStack

```
class IntStack {  
    // .../  
public:  
    IntStack (int);  
};  
void foo (IntStack& i) {  
    i.Push(0);  
}  
int main () {  
    foo (10);  
    IntStack stack ('a');  
}
```

You **don't** want a **user** to do such a thing: use **explicit**

# Conversion Operator

```
void foo (int i) {}  
int main () {  
    Integer i (12);  
    Integer j (50);  
    i < j;           // ok if you define Integer::operator<  
    100 < j;         // ok if operator< is as a global friend function  
    foo (i);         // error but you may need it  
// File.C: In function 'int main()':  
// File.C:61: 'class Integer' used where a 'int' was expected  
}
```

In C++, another **way** to **provides automatic type** conversion is through **conversion operator**

# Conversion Operator

```
class Integer {  
    // .../  
    operator int () const {  
        return integer;  
    }  
};  
void foo (int a);  
int main () {  
    Integer i (12);  
    foo (i);  
}
```

You **don't** need to specify **return type**: it is the **name** of the **declared operator**



# Ambiguity between Conversion and Constructor

```
#include <iostream>
using namespace std;
class Y;
class X {
public:
    operator Y ();
};
class Y {
public:
    Y () { cout << "Y::Y()\n"; }
    Y (X) { cout << "Y::Y(X)\n"; }
};
```

```
X::operator Y () {
    cout << "X::operator Y()\n";
    return Y();
}
void foo (Y) {}
int main () {
    X x;
    foo (x);
}
```

**two ways** to do the **transtyping** : the **transtyping constructor** `Y::Y(X)`  
and the conversion operator `X::operator Y()`

# Operator Conversion and Overloading

```
class A {};  
class B {};  
class C {  
    operator A () const;  
    operator B () const;  
};  
void foo (A);  
void foo (B);  
int main () {  
    C c;  
    foo (c);  
}
```

foo (c.operator A()) or foo (c.operator B()) : C++ **cannot choose** it raises a **compile-time error**

# Pass Large objects by Reference

```
const Integer operator+ (const Integer arg1, const Integer arg2);
```

Copying two integers is neglectable and probably acceptable

```
const Matrix operator+ (const Matrix& m1, const Matrix& m2) {  
    Matrix sum;  
    // .../  
    return sum;  
}
```

For **Large objects** pass arguments **by reference**

**Reference** allows the use of **expressions** involving **usual arithmetic operator without copying**

# The subscripting operator []

It must be a **member function**

It requires a **single argument**

If you want your object to **act like** an **array** (left and right value): the **subscript** operator must return a **reference**

```
class IntArray {  
    int size;  
    int* array;  
public:  
    int& operator[] (int i) { return array [i]; }  
};
```

The second argument of an operator [] function may be of any type

# Function call operator

Must be a **member function**

```
class Integer {  
public:  
    int operator() (int i, int j = 0) {  
        integer += i+= j;  
        return integer;  
    }  
    int operator() (int i, int j, int k) {  
        integer += i+= j += k;  
        return integer;  
    }  
};  
int main () {  
    Integer i = 12;  
    cout << i(12) << endl;  
    cout << i(12, 14) << endl;  
    cout << i(12, 14, 32) << endl;  
}
```

# The comma operator

Called when a **comma** appear **after** the **object** for **which** it is **defined**  
but **not called** for **function argument list**

```
class Integer {  
public:  
    Integer (int i);  
    Integer& operator, (const Integer arg) {  
        cout << "what do you want such an operator to do ?";  
        return *this;  
    }  
};  
int main () {  
    Integer i = 12, j = 10, k = 23;  
    i, j, k, 2;  
}
```

Do not use this operator : it exists for language consistency

## 27. new / delete

# The Free Store

Useful to **create object** that **exists independantly** of the **scope** in which it was created

```
IntStack* foo () {  
    return new IntStack (100);  
}
```

The **IntStack** allocated inside **foo** is returned and can be used outside the function

Free store operators **new**, **delete**, **new []**, **delete []** are implemented:

```
void* operator new (size_t);  
void operator delete (void*);  
void* operator new[] (size_t);  
void operator delete[] (void*);
```

The **new** and **new[]** on **primitive types** do **not initialize** the returned memory



# Memory Exhaustion: the **new\_handler** solution

When the **operator new** cannot find **contiguous** block of storage to hold the desired object it **fails**

When operator **new** fails it calls the function you have specified with **std::set\_new\_handler**

It checks if the **new\_handler** pointer to function exists and if yes it calls it

The **new\_handler** default behavior is to **throw** an **exception**

# Memory Exhaustion : the **new\_handler** solution

```
#include <iostream>
using namespace std;

void memory_exhaustion () {
    cerr << "Memory Exhaustion";
    exit(1);
}

int main () {
    set_new_handler(memory_exhaustion);
    for (;;) new int [100000];
}
```

Memory Exhaustion

# Memory Exhaustion : the exception solution

When **new** cannot find **space** to **allocate** by default : it **throws** the **std::bad\_alloc** exception

```
#include <iostream>
#include <new>
using namespace std;
void foo () {
    try {
        for (;;) new int[100000];
    }
    catch (bad_alloc) {
        cerr << "Memory Exhaustion\n";
    }
}
int main () {foo();} // Memory Exhaustion
```

# The exception solution

Memory Exhaustion throws the exception: `std::bad_alloc`, you can catch it

But `new_handler` is called before it

```
#include <iostream>
#include <new>
void out_of_store () {
    cerr << "Out of Store\n";
    exit(1);
}
void foo () {
    set_new_handler(out_of_store);
    try {
        for (;;) new int[100000];
    }
    catch (bad_alloc) {
        cerr << "Memory Exhaustion\n";
    }
}
int main () {foo();}
Out of Store
```

# The `std::nothrow` solution

You can tell the **operator new** to return **0** instead of **throwing exception** or calling **new\_handler**

```
#include <iostream>
#include <new>
using namespace std;

int main () {
    int* exhaustion = 0;
    while (!exhaustion)
        exhaustion = new(nothrow) int [100000];
    cout << "Allocation failed" << endl;
}
```

Allocation failed

# Overloading new and delete

it is **possible** to **overload** the **global new** and **delete** operators

**BUT** do not forget that **someone else** might **rely** on the new and delete **default behavior**

**prefer** supply these operators for **specific classes**

**remember** when **overloading new** : the **first** argument is always the **size** of the **memory** to be allocated

# new in Pre-Allocated Memory

```
#include <iostream>
#include <new>
using namespace std;
class X {
public:
    X () { cout << "X::X\n"; }      ~X () { cout << "X::~X\n"; }
    static void* operator new (size_t size, void* location) {
        cout << "X::operator new" << endl;
        return location;
    }
    static void* operator new (size_t size) {
        cout << "new malloc" << endl;
        return malloc(size);
    }
};

int main () {
    // pre allocated buffer for new
    int tab [100];
    X* px1 = new (tab) X ();
    X* px2 = new X();
    px1->X::~X();
    free(px2);}

// X::operator new
// X::X
// new malloc
// X::X
// X::~X
```

# Overloading delete ?

```
class X {
public:
    static void* operator new (size_t size, void* location) {
        cout << "X::operator new" << endl;
        return location;
    }
    static void* operator new (size_t size) {
        return malloc(size);
    }
    static void operator delete (void* p, void* location) {
        // nothing to do
        cout << "X::operator delete" << endl;
    }
    static void operator delete (void* p) {
        free(p);  }
};
```

```
int main () {
    int tab [100];
    X* px1 = new (tab) X ();
    X* px2 = new X();
    delete px1; // Segmentation fault (core dumped)
    delete px2;
}
```



# Overloading new and delete

**Notice** that you have to call **explicitly** the **destructor** of X

you can implement a delete operator with arguments

BUT you can never pass arguments to delete

no way to define a symmetric operator delete in place !

## 28. TP3 : implement the String class

# Operator Overloading : the MyString class

- implement a class MyString to store character string elements
- use the basic functions of the `<cstring>` file
- implement the operator `+` for the string concatenation
- implement the subscript operator to access a character of the string
- implement conversion to character string
- create an array of MyString and sort it using your bubble sort

## 29. TP3 : Overloading Operators

# Overloading new and delete

the new operator returns a piece of memory from a pre-allocated buffer

## 30. Derived Class

# Derived Classes

A class **implements** a **user-defined** type with a set of **data** and **behaviors** through class **data members** and class **member functions**

Sometimes, the **set** of members becomes **insufficient** and you need to **add** more **data** and **behaviors** to some code

How will you extend a code if:

You **don't own** the **code**, it comes from a **library**

You don't want to touch the **code**, it is already **developped** and **debugged**

# Extend an Existing Code

The **first way** to do this in C++ is with **composition**:

Embed an **object** of the existing type in your new **class** as a **data member** (the new class accesses the object's public interface)

The **second way** to do this is with **inheritance**:

Create a **new class** as a **sub-type** of the **existing** class (the **new** class contains the **public** and **protected** part of the **existing** class)



# Embed Objects in your Class with Composition

```
class Engine {};  
class Wheel {};  
class Window {};  
class Door {};  
class Vehicle {  
    Engine engine;  
    Wheel wheels[4];  
    Door doors [2];  
};
```

# Composition

Make **objects private** to **prevent** the outside from **changing** it without your **control**

The **object** becomes a **part** of the **underlying implementation** of your **class**

If you only want to **use** the **features** of **existing classes** inside a **new class** : use **composition**

# Composition or Inheritance ?

```
class Car { Vehicle vehicle; };  
class Truck { Vehicle vehicle; };
```

Car and Truck are both composed with Vehicle but :

- it is not possible to connect a Car or a Truck to a Vehicle
- there is no way to have an heterogenous container mixing Car and Truck

A Car (resp. a Truck) does **not contain** a Vehicle : **it is** a Vehicle

```
class Vehicle {};  
class Car : public Vehicle {};  
class Truck : public Vehicle {};
```

To **hide** implementation details to the **outside** but **not** to **derived** classes : qualify a member with **protected**

# Initialization List

```
class Vehicle {  
public:  
    Vehicle (int n) {}  
};  
class Car : public Vehicle {  
public:  
    Car (int n) : Vehicle(n) {}  
};
```

The initialization list is the only place to call base class constructor(s) with arguments

# Upcasting is Safe

if a **derived class** has a **public base class** :

an **object** of the **derived** class can be **treated** as an object of its **base** class

when **manipulated** through **pointers** or **references**

the **opposite** is not **true** (use `static_cast` and `dynamic_cast`)

```
class Base {};  
class Derived : public Base {};  
int main () {  
    // upcasting  
    Derived* d1 = new Derived();  
    Base* b1 = d1;  
    // ok an object of type Derived is also an object of type Base  
    // downcasting  
    Base* b2 = new Derived();  
    Derived* d2 = (Derived*)b2;  
    // ok b2 is of type Derived  
    Base* b3 = new Base();  
    Derived* d3 = (Derived*)b3;  
    // FALSE but ok for the compiler  
}
```

# Issue when Upcasting

```
struct Shape {
    int x, y; //<<< Position of the shape
    void move(int dx, int dy) {
        x+=dx; y+=dy;
    }
    float area() const;
};

struct Circle : public Shape {
    int radius;
    float area() const;
};

struct Rectangle : public Shape {
    float area() const;
};

int foo () {
    Circle * c = new Circle();
    Shape * s = c;    // Upcasting
    c->move(100,100); // Call Shape::move(...)
    s->move(100,100); // Call Shape::move(...)
    c->area();        // Call Circle::area()
    s->area();        // /\ Call Shape::area() /\
}
```

# Base Class

Circle and Rectangle are two **graphic shapes**

To represent Circle and Rectangle without losing their **common notion** of Shape, we derive the two classes from the same base class Shape

For **some** functions like `move(int, int)`, the **base** class data members are **enough** to **give** the **implementation** : the **behavior** does not depend on the **object type**

For some **other** functions like `area`, we **cannot find** a **common implementation** : the **behavior** depend on the **object type**

# Upcasting and Static Binding

**Treating** a **pointer** to an **object** (Circle) as being a **pointer** to its **base class** type Shape is **correct**, **frequently used**

The function **call** is connected to the function **body** by **static binbing**

The area called is Shape::area not Circle::area

**Problem:**this is **clearly not** the **desired** behavior



# Solving the upcasting issue first solution

To fix the issue of static binding we can use **function pointers**.

```
void foo(int a, int b, float c) { /*...*/ }
int main() {
    // define a variable that is pointer to a
    // function
    void (*func_pointer0) (int, int, float)
        = &foo;
    // Use the function pointer:
    func_pointer0(10, 20, 10.0f);
    (*func_pointer0)(10, 20, 10.0f);
}
```

We can add a function pointer to the shape class and replace it when we use it with circle.

# Solving the upcasting issue first solution

```
struct Shape {
    int x, y; //<<< Position of the shape
    float (*varea) (Shape const *);
    Shape() : x(0), y(0), { varea = &Shape::_area; }
    float area() { return *varea(this); } // Call varea
    static float _area(Shape const * ths) { /*...*/ }
};

struct Circle : public Shape {
    int radius;
    Circle() { varea = &Circle::_area; } // Replace the function pointer
    float area() const;
    static float _area(Shape const * ths) { /*...*/ }
};

int foo () {
    Circle * c = new Circle();
    Shape * s = c;    // Upcasting
    c->area();         // Call Circle::area()
    s->area();         // Call Shape::area() that is bound to Circle::_area
}
```

This solution is not automatic, not optimal and complex.

This is the solution used in C.

# Upcasting and Dynamic Binding: virtual keyword

C++ **provides** a **mechanism** to **determine** the type of an **object** at **run time** : **dynamic binding**

Such a **mechanism** is **done** with the **virtual keyword**

**Cause dynamic binding** to **occur** for a **function** by declaring that **function virtual** (default is static binding)

# Problem with Static Binding when Upcasting

```
class Shape {
protected:
    int x, y;
public:
    void move (int dx, int dy) {
        x+=dx;
        y+=dy;
    }
    virtual float area () const {
        cout << "Shape::area";
    }
};

class Circle : public Shape {
protected:
    int radius;
public:
    float area () const {
        cout << "Circle::area";
    }
};

void foo (Shape* s) {
    s->area();
}

int main () {
    Circle* c = new Circle();
    foo(c);
}

// Circle::area
```

# Problem if you Don't Use Virtual Destructor

```
class A {  
public:  
    A () { cout << "A::A\n"; }  
    ~A () { cout << "A::~~A\n"; }  
};  
class B : public A {  
public:  
    B () { cout << "B::B\n"; }  
    ~B () { cout << "B::~~B\n"; }  
};  
int main () {  
    A* pa = new B();  
    delete pa;  
}  
// A::A  
// B::B  
// A::~~A
```

# Use Virtual Destructor !

Because **objects** of **derived** classes are **often** manipulated and **deleted** through **pointer** to the **base class** : supply a **virtual destructor** in base classes

```
class A {  
public:  
    A () { cout << "A::A\n"; }  
    virtual ~A () { cout << "A::~~A\n"; }  
};  
class B : public A {  
public:  
    B () { cout << "B::B\n"; }  
    ~B () { cout << "B::~~B\n"; }  
};  
int main () {  
    A* pa = new B();  
    delete pa;  
}  
// A::A  
// B::B  
// B::~~B  
// A::~~A
```

The destructor of A is implemented only to manipulate pointers on derived classes through pointer on A

# Order of Constructor and Destructor for Inheritance

The **construction**:

**starts** at the **very base class hierarchy**

at **each level**, **object data member constructors** are **called**

**followed** by the call of the **current constructor**

**Destructors** are called in **exactly** the **reverse order** of **constructors**

```
class A {
public:
    A () { cout << "A::A\n"; }
    ~A () { cout << "A::~A\n"; }
};
class B {
public:
    B () { cout << "B::B\n"; }
    ~B () { cout << "B::~B\n"; }
};
class C : public B {
public:
    C () { cout << "C::C\n"; }
    ~C () { cout << "C::~C\n"; }
};
class D : public C {
public:
    A d;
    D () { cout << "D::D\n"; }
    ~D () { cout << "D::~D\n"; }
};
int main () {
    D d;
}
// B::B => C::C => A::A => D::D
// D::~D => A::~A => C::~C => B::~B
```



# Derivation and Copy Constructor **X (const X&)**

If the **derived class** does **not define** a **copy constructor** instance:

- Default memberwise initialization is applied whenever one object of the derived class is initialized with another
- The copy constructor is invoked for each base class for which it is defined

If the derived class does define a copy constructor instance

- It is a constructor, so handling of the base class parts becomes its responsibility !

```
class A {
public:
    A () { cout << "A::A\n"; }
    A (const A&) { cout << "A::A(constA&)\n"; }
};
class B {
public:
    B () { cout << "B::B\n"; }
    B (const B&) { cout << "B::B(constB&)\n"; }
};
class C {
public:
    C () { cout << "C::C\n"; }
    C (const C&) { cout << "C::C(constC&)\n"; }
};
class D : public A, public B {
public:
    C c;
    D () { cout << "D::D\n"; }
};
int main () {
    D d1;
    D d2 = d1;
}
// A::A => B::B => C::C => D::D
// A::A(constA&) => B::B(constB&) => C::C(constC&)
```

For d2, C++ calls A, B and C **copy constructors** and D default copy constructor

```
class A {
public:
    A () { cout << "A::A\n"; }
    A (const A&) { cout << "A::A(constA&)\n"; }
};
class B {
public:
    B () { cout << "B::B\n"; }
    B (const B&) { cout << "B::B(constB&)\n"; }
};
class C {
public:
    C () { cout << "C::C\n"; }
    C (const C&) { cout << "C::C(constC&)\n"; }
};
class D : public A, public B {
public:
    C c;
    D () { cout << "D::D\n"; }
    D (const D&) { cout << "D::D(constD&)\n"; }
};
int main () {
    D d1;
    D d2 = d1;
}
// A::A => B::B => C::C => D::D
// A::A => B::B => C::C => D::D(constD&)
```

D::D(const D&) calls implicitly the **default constructors** of the base classes A, B and C

**NOT** the **copy constructors** of the base classes A, B and C

```
class A {
public:
    A () { cout << "A::A\n"; }
    A (const A&) { cout << "A::A(constA&)\n"; }
};
class B {
public:
    B () { cout << "B::B\n"; }
    B (const B&) { cout << "B::B(constB&)\n"; }
};
class C {
public:
    C () { cout << "C::C\n"; }
    C (const C&) { cout << "C::C(constC&)\n"; }
};
class D : public A, public B {
public:
    C c;
    D () { cout << "D::D\n"; }
    D (const D& d) : A (d), B (d), c(d.c) { cout << "D::D(constD&)\n"; }
};
int main () {
    D d1;
    D d2 = d1;
}
// A::A => B::B => C::C => D::D
// A::A(constA&) => B::B(constB&) => C::C(constC&) => D::D(constD&)
```

`D::D(const D&)` is responsible of all calls to base classes and composed classes copy constructors of A, B and C

# Base Class Abstract or Concrete ?

A **base** class may represent a **concrete** implementation of some concept

It can then be **useful** to create a base class **object**

When a base class is a concrete class, you can give a concrete implementation to all the member functions

# Function with No Meaning on a Base Class Object

A **base** class may **represent** an **abstract concept** for which **object** do not make sense

```
class Shape {  
    int x, y;  
public: void move (int dx, int dy) {...}  
        virtual void area() const {}  
};  
class Circle : public Shape {  
public: void area () const {...}  
};
```

When a base class is an **abstract concept**, you cannot **provide** correct **implementations** for all its **virtual** functions

**Shape::move** has a meaning but **Shape::area** does not

# Example of Abstract Class

```
class Shape {  
protected:  
    int x, y;  
public:  
    void move (int dx, int dy) {  
        x+=dx;  
        y+=dy; }  
    virtual float area () const {}  
};  
int main () {  
    Shape* s = new Shape();  
    s->move(10, 10);  
    s->area(); }
```

It is **Legal** to **create** a Shape object and to call **member functions** move and area but it has no meaning

**Problem** : create a Shape is a **bad use** of your **code** and you want to forbid the user from doing it

# Pure Virtual Functions

**Shift this problem to the compiler** by declaring **pure virtual functions** :  
the body of a **pure virtual functions** is replace by 0

```
class Shape {
protected:
    int x, y;
public:
    void move (int dx, int dy) {
        x+=dx;
        y+=dy; }
    virtual float area () const = 0;
};

int main () {
    Shape* s = new Shape(); // line 14
    s->move(10, 10);
    s->area(); }

// : In function 'int main()':
// :14: cannot allocate an object
//       of type 'Shape'
// :14:   since the following
//       virtual functions are abstract:
// :14:   float Shape::area() const
```

The **compiler** knows that Shape is **abstract** : it contains a **pure member function**  
the **creation** of a Shape raises a **compile time error**



# virtual Function

```
class Shape {  
protected:  
    int x, y;  
public:  
    void move (int dx, int dy) {  
        cout << "Shape::move\n";  
        x+=dx;  
        y+=dy;  
    }  
    virtual float area () const = 0;  
};
```

```
class Circle : public Shape {  
public:  
    virtual float area () const {  
        cout << "Circle::area\n";  
    }  
};  
  
int main () {  
    Circle* s = new Circle();  
    s->move(10, 10);  
    s->area();  
}
```

# Pure Virtual Functions

A **pure virtual function** that is **not defined** in a derived class remains a **pure virtual** function

The **derived classes** that do not define pure virtual functions become **abstract classes**

If a class is **abstract** you **cannot create** an **object** or **allocate** one with **new** but you can **use** it as **pointer** and **reference**

```
void foo (Shape* pShape) {  
    Shape* pCircle = new Circle ();  
    if (pShape->area() == pCircle->area()) {}  
}
```

# Provide an Interface

An important use is to **provide** an **interface** without knowing any implementation details and specify the **implementation** in **derived class**

```
class CharacterDevice {  
public:  
    virtual int open () = 0;  
    virtual int close (const char*) = 0;  
    virtual int read (const char*, int) = 0;  
    virtual int write (const char*, int) = 0;  
    ...  
};
```

And specify the actual device driver as classes derived from **CharacterDevice**

# Grouping Exceptions : use Inheritance

```
class MathError {};  
class Overflow : public MathError {};  
class Underflow : public MathError {};  
class Zerodivide : public MathError {};  
  
void foo () {  
    try { ... }  
    catch (Overflow) {  
        // handle Overflow or anything derived from Overflow  
    }  
    catch (MathError m) {  
        // general case : handle any MathError except Overflow  
    }  
}
```

# Inheritance

The **base class** is **preceded** by the **public** keyword

We qualify the inheritance with the **public** keyword because by default it is **private**

# Functions That Are Not Inherited

all member **functions** are **NOT automatically inherited** from **base** class to **derived** class

All the **constructors** and **destructors** of the whole hierarchy are **called** when an **object** is **created** and **destroyed**: they are **not inherited**

The **copy constructor** and the **operator=** are **synthetised** by the compiler if you do not implement them so they are **not inherited**

But the **automatic type conversion** is **inherited**: a derived class object is also a base class object and can be converted in the same way

# Member Access Control

A member of a **class** can be:

**private:** the name can be used only by **member functions** and **friends** of the class in which it is declared

**protected:** the name can be used only by **member functions** and **friends** of the class in which it is declared and by **member functions** and **friends** of the classes **derived** from this class

**public:** the name can be used everywhere

# Access Specifiers When Deriving From a Base Class

You declare a class to be a base class by using **access specifiers**: **public**, **protected**, and **private** (default)

**public:** **public** members of the base class are **public** members of the derived **class** and **protected** are **protected**

**protected:** **public** and **protected** members of the base class are **protected** members of the derived **class**

**private:** **public** and **protected** members of the base class are **private** members of the derived class



# Access to Base Classes : Same as Access to Members

**Base** is a **public** base class of **Derived1**: any **function** can implicitly convert an **Derived1\*** to an **Base\*** where needed

**Base** is a **protected** base class of **Derived2**: only **member** and **friends** of **Derived2** and **member** and **friends** of **Derived2**'s derived classes can (implicitly) convert an **Derived2\*** to an **Base\*** where needed

**Base** is a **private** base class of **Derived3**, only **member** and **friends** of **Derived3** can (implicitly) convert an **Derived3\*** to a **Base\*** where needed.

```
class Base {};  
class Derived : private Base {  
    void foo () {  
        Base* p = new Derived();  
    }  
};  
int main () {  
    Base* p = new Derived();  
    // compile-time error :  
    // 'Base' is an inaccessible base of 'Derived'  
}
```

# Private Inheritance

**Private inheritance** is useful: it behaves like “**reimplementing in term of**”

You use **data** and **functionality** of the base class but they are **hidden**

A **user cannot treat** your **object** as a **base class member** object

**Derive private** an **integer array** to build an **integer stack**: you have restricted the outside to access every where in your stack

```
class IntArray {
    int* array;
    int size;
public:
    IntArray (int);
    int& operator [] (int);
};

class IntStack : private IntArray {
public:
    IntStack (int);
    void Push (int);
    int Pop ();
};

void foo () {
    IntStack stack (100);
    stack.Push(10);
    stack[1] = 12;
    // 'operator []' is from private base class
}
```

# inheritance : the expression class hierarchy

implement classes to store, evaluate and print arithmetic expressions such as  $2$ ,  $4 + 5$ ,  $8 * 6$ ,  $2 + 4 * 7 - 12$   
you do not have to parse the expression just to design the class hierarchy to store the expression

## 33. Multiple Inheritance

# Class Hierarchy

a class can have more than one direct base class

```
class Name {  
    //...  
public:  
    void changeFont ();  
    void draw ();  
};  
class Vertex {  
    //...  
public:  
    int numberOfEdges ();  
    void draw ();  
};
```

```
class NamedVertex :  
    public Vertex, public Name {  
public:  
    void draw ();  
};  
void foo(NamedVertex* v) {  
    v->changeFont(); // Name::changeFont  
    v->numberOfEdges(); // Vertex::numberOfEdges  
}
```


# Multiple Inheritance Ambiguity

Access to **base class members** is **ambiguous** if the expression used refers to more than one function or data

```

class A {
public:
    int a;
    void f ();
};
class B : public A { };
class C : public A { };
class D : public B, public C {
    void g ();
};
void D::g () {
    a = 3;           // error: ambiguous A::a and A::a
    B::a = 2;        // ok : a from the A part of B
    f ();            // error: ambiguous A::f () or A::f ()
    D* d = new D;
    A* pa = d;       // error: cast: D* -> base A*; A is base more than once
    pa = (A*)(C*) d; // ok A part of C
    pa = (B*) d;     // ok A part of B
}

```



# Virtual Inheritance

To **avoid** having **two** sub-objects of **same type** in **multiple inheritance**:  
**inherit virtual**

```
class V {
public:
    int a;
    void f ();
};
class B : virtual public V { };
class C : virtual public V { };
class D : public B, public C {
    void g ();
};
void D::g () {
    a = 3;           // ok : only one V::a
    f ();           // ok : only one V::f()
    D* d = new D;
    V* pa = d;      // ok: only one base class V in D
}
```



**C** has only **one sub-object** of class **V**



# Virtual Inheritance

**constructors** of all **virtual base classes** are **invoked** (implicitly or explicitly) from the constructor for the **complete object**

**Any class**, no matter how far away it is from the **virtual base class**, is **responsible** for **initializing** the **base class**

```
class Base {  
public:  
    Base (int i) {}  
};  
class Derived1 : virtual public Base {  
public:  
    Derived1 () : Base (1) {}  
};
```

# 34. Template

# Templates

for (containers) classes (**list**, **set**, **array**, ...), the type of contained objects is of little interest

**C++** allows a **type** to be a **parameter of a class**

a keyword (**template**) indicates that a class manipulates non-specified types

programmer implements a class using parameters instead of some precise types

users instantiation types when needed

# Stack Template (functions are defined inside the class)

```
#include <iostream>
using namespace std;

template <class T>
class Stack {
    int top;
    int size;
    T* vector;
public:
    Stack (int s = 10) : top(0), size (s), vector (new T [size]) {}

    T& operator[] (int index) {
        if ((0 <= index) && (index < getSize())) {
            return vector[index];
        }
    }

    void push (T t) { vector[top++] = t; }
    T pop () { return vector[--top]; }
    int getSize () const { return size; }
    int nbOfElems () const { return top; }
};
```

# Stack Template (functions are defined inside the class)

```
template <class T>
ostream& operator<< (ostream& os, const Stack<T>& tStack) {
    os << "[";
    for (int i = 0; i < tStack.nbOfElems(); ++i) {
        os << tStack[i] << " ";
    }
    os << "]";
    return os;
}

int main () {
    Stack<int> intStack;
    for (int i = 0; i < 10; ++i) {
        intStack.push(i);
    }
    cout << intStack.pop() << endl;
    cout << intStack << endl;
    intStack.push(1);
    Stack<double> doubleStack;
}
```

# Template Stack

```
#include <iostream>
template <class T>
class Stack {
    int top;
    int size;
    T* vector;
public:
    Stack (int s = 10);
    T& operator[] (int index);
    void push (T t);
    T pop ();
    int getSize () const;
    int nbOfElems () const;
};

template <class T> inline int Stack<T>::getSize () const { return size; }
template <class T> inline int Stack<T>::nbOfElems () const { return top; }
template <class T> inline Stack<T>::Stack (int s = 10) :
    top(0), size(s), vector (new T [size]) {}
template <class T> inline T& Stack<T>::operator[]
    (int index) {
    if (0 <= index < getSize()) {
        return vector[index];
    }
}

template <class T> inline void Stack<T>::push (T t) { vector[top++] = t; }
template <class T> inline T Stack<T>::pop () { return vector[--top]; }
```

# Template Stack

```
int main () {  
    Stack<int> intStack;  
    for (int i = 0; i < 10; ++i) {  
        intStack.push(i);  
    }  
    cout << intStack.pop() << endl;  
    cout << intStack << endl;  
    intStack.push(1);  
    Stack<double> doubleStack;  
}
```

```

#include <iostream>
template <class T>
class Stack {
    int top;
    int size;
    T* vector;
public:
    Stack (int s = 10);
    ~Stack ();
    T& operator[] (int index);
    void push (T t);
    T pop ();
    int getSize () const;
    int nbOfElems () const;
};

template <class T>
inline Stack<T>::~~Stack () { delete [] vector; }

// .cpp file
#include "StackTemplate.h"
class Shape;
class String;
class Date {};
Stack<Shape*>  SHAPE_STACK (200);
Stack<String*> STR_STACK  (100);

void foo (Stack<Date>& dateStack) {
    dateStack.push(Date());
    Stack<int*> pIntStack = NULL;
    pIntStack = new Stack<int> (10);
    pIntStack->push(1);
}

```



# Instances of Template Functions

The compiler ensures that all the needed versions of template functions are generated

Here :

- constructor for  
`Stack<Shape*>`, `Stack<String*>`,  
`Stack<int>`
- **push ()** function of  
`Stack<Date>` and  
`Stack<int>`
- Destructor for  
`Stack<Shape*>`,  
`Stack<String*>`

# How to use template in Practice.

Put in a **header** file

- the entire template **declaration**
- the entire template **definition**

Template definitions are **special** :

- the compiler do not generate any thing until instantiation
- you do not have to prevent multiple definition (the compiler does it for you) // not true

# template function

You can declare template function

```
template <class T> void Sort (Vector<T>& v) {  
    ...  
    if (v[i] < v[j]) {  
        ...  
    }  
    void foo (Vector<int>& vInt) {  
        Sort(vInt);  
        Sort<int>(vInt);  
    }  
}
```

# template function

You can specialize template function if needed

```
// because < on char* does not exist
void Sort (Vector<char*>& vChar) {
...
    if (strcmp(v[i], v[j]) < 0) { ...
...
}
void bar (Vector<char*>& vChar) {
    Sort(vChar);
}
```

# Template Argument

Template arguments are not restricted to type

you can declare compile-time constants

```
template <class T, const int CONST_SIZE>
class Stack {
    int top;
    int size;
    T vector [CONST_SIZE];
public:
    Stack () : top(0), size (CONST_SIZE) {}
    // ...
};
```

# Template Argument

```
#include <iostream>
template <class T, const int CONST_SIZE>
class Stack {
    int top;
    int size;
    T vector [CONST_SIZE];
public:
    Stack () : top(0), size (CONST_SIZE) {}
    T& operator[] (int index) {
        if (0 <= index < getSize()) {
            return vector[index];
        }
    }
    void push (T t) { vector[top++] = t; }
    T pop () { return vector[--top]; }
    int getSize () const { return size; }
    int nbOfElems () const { return top; }
};
```

# Template argument

```
template <class T, const int CONST_SIZE>
ostream& operator<< (ostream& os, Stack<T, CONST_SIZE>& tStack) {
    os << "[";
    if (tStack.nbOfElems() > 0) {
        for (int i = 0; i < tStack.nbOfElems(); ++i) {
            os << tStack[i] << " ";
        }
    }
    os << "]";
    return os;
}

int main () {
    Stack<int, 1000> intStack;
    cout << intStack.getSize() << endl;
    for (int i = 0; i < intStack.getSize(); ++i) {
        intStack.push(i);
    }
    cout << intStack << endl;
    cout << intStack.pop() << endl;
    cout << intStack << endl;
    intStack.push(1);
    Stack<double, 20> doubleStack;
}
```

# typename

the **typename** keyword indicates that an identifier inside a template is a type

```
template <typename T>
class MyClass {
    T::field * t;           // SYNTAX ERROR
};
```

```
template <typename T>
class MyClass {
    typename T::field * t;  // OK
};
```

```
template <typename T>
class MyClass {
    typedef typename T::field TFIELD;
    TFIELD* t;
};
```



# Nontype Function Template Parameters

```
#include <iostream>
template <typename T, int VALUE>
T addValue (const T& x) {
    return x + VALUE;
}
int main () {
    int i = 100;
    std::cout << addValue<int, 5>(i) << std::endl;
}
```

# Controlling instantiation ?

Sometime, it it is useful to explicitly instanciate a template even when you don't use it

# Explicit Instantiation

```
template <class T>
class Stack {
    T* vector;
public:
    Stack (int size) :
        vector (new T [size]) {}
};
```

```
class A {
public:
    A (int i) {}
};
template class Stack<A>;
```

ERROR : to create an array of T, T needs  
to have a default constructor

# Static Template Class Members

You can declare static template members

```
template <typename T>
class A {
public:
    static T value;
};
```

you must define them before their use

```
template <typename T>
T A<T>::value = 0;
int main () {
    A<int>::value = 0;
}
```

# Explicit Instantiation Directive

the **explicit instantiation** directive is a construct to instantiate (explicitly defined) templates manually

the keyword **template** is followed by the fully substitution declaration of the entity we want to instantiate

# Function and Class Explicit Instantiation

## explicit instantiation of a function template

```
template <typename T>  
T foo (T) {}
```

```
template int foo<int>(int);
```

## explicit instantiation of a class template

```
template <typename T>  
class A {  
public:  
    template <typename T2>  
    class B {};  
};
```

```
template class A<int>::B<float>;
```

# Member Function and Static Member Explicit Instantiation

explicit instantiation of a member function (constructor)

```
template <typename T>
class A {
public:  A() {}
};
```

```
template A<double>::A();
```

explicit instantiation of a static data member

```
template <typename T>
class A {
    static T staticT;
};
template <typename T>
T A<T>::staticT;
```

```
template float A<float>::staticT;
```

# Explicit Instantiation for Using Template in Practice ?

each directive must appear only once in a program

not following this rule results in linker errors

explicit instantiation has a clear disadvantage : you must keep track of which entities to instantiate

for large projects this becomes excessive burden

but you avoid the overhead of large headers

the source code of template can be kept hidden

no addition instantiations can be created by a client program



# Template Function in Template Class

```
template <class T>
class A {
public:
    template <class S> void foo (S&);
    template <class S> void bar (A<S>& s) {}
};

template <class T> template <class S>
void A<T>::foo (S& s) {}
```

# Specialization of Class Template

you can specialize a class template for certain template arguments

```
template <typename T>
class MyClass {
public:
    T t;
    void myFunction () {}
};
```

```
void MyClass<char*>::myFunction () {}
int main () {
    MyClass<int> c1;
    c1.myFunction();
    MyClass<char*> c2;
    c2.myFunction();
}
```

# Partial Specialization

class templates can be **partially specialized** for a family or template arguments

```
template <typename T1, typename T2>
class MyClass {
public:
    T1* t1;
    T2* t2;
};
```

# Partial Specialization

```
template <typename T>
class MyClass <T, T> {
public:
    T* t;
};
```

```
template <typename T>
class MyClass <T, double> {
public:
    T* t;
};
```

```
template <typename T1, typename T2>
class MyClass <T1*, T2*> {
public:
    T1* t;
};
```

```
int main () {
    MyClass<int, char> c1;        // <T1, T2>
    MyClass<int, int> c2;        // <T, T>
    MyClass<int, double> c3;     // <T, double>
    MyClass<int*, char*> c4;     // <T1*, T2*>
}
```

# The Angle Bracket Hack

sometime you need to add some blank space between closing brackets

>> constitutes a right-shift operator even when it makes no sense at that location

```
template <typename T> class A {};  
template <typename T> class B {};  
int main () {  
    B<A<int>>> ba1;    // ERROR  
    B<A<int> > ba2;    // OK  
}
```

# Exception for class **template**

We have the choice when naming an exception

- Each generated class can have its own exception class
- An exception can be common to all classes generated from the **templates**

```
template <class T> class Vector {  
    class RangeError {};  
    ...  
};  
void foo (Vector<int>& vi, Vector<Complex>& vc) {  
    try { ...}  
    catch (Vector<int>::RangeError) { ...}  
    catch (Vector<Complex>::RangeError) { ...}  
}
```

```
class RangeError {...};  
template <class T> class Vector {  
    ...  
};  
void foo (Vector<int>& vi, Vector<Complex>& vc) {  
    try { ...}  
    catch (RangeError) { ...}  
}
```

transform your `IntStack` or your `MyList` class in a template class



## 36. Input Output Streams

# Basic Input and Output Operations

standard input and output library are used to print messages on the screen and get input from the keyboard

to perform input and output operations, a program does not need to know exactly the physical location (screen, keyboard)

it only needs to know that the location will accept or provide characters in a sequential way

the program performs its operations on a `stream` (an abstraction of the physical location)

# Input and Output in C++

C++ contains a stream library (a stream formats and holds bytes)

You can have **input** stream: **istream** and **output** stream: **ostream**

**ifstream** and **ofstream** handle **files I/O**

**istrstream** and **ostrstream** handle **string I/O**

**istream** and **ostream** are the interface with the C++ standard library **String**

Three standard streams are predefined whenever you include `<iostream>`:  
**cin** (standard input stream), **cout** (standard output stream) and **cerr** (standard error stream)

# Overloading Operators for Input and Output

An uniform traitement for built-in types and user defined types is archieved by overloading the two operators >> for input and << for output

Functions are offered for input and Output of Built-in Type

The operator >> function skips whitespace before reading

```
int i;          cin >> i;
float f;        cin >> f;
char buff[100]; cin >> buf;

cout << "int i = "          << i    << endl;
cout << "float f = "        << f    << endl;
cout << "char buff[100] = " << buf  << endl;
```

The **space** is the delimiter of inputs

# Using Global Functions for Output of User Defined Type

```
#include <iostream>
class A {
friend
    ostream& operator<< (ostream& os, A&);
private:
    int a;
public:
    A (int i) : a(i) {}
};
inline ostream& operator<< (ostream& os,
                           A& ra) {
    os << "A::a=" << ra.a;
    return os;
}
int main () {
    A a1 (12);
    A a2 (13);
    A a3 (14);
    cout << a1 << " "
         << a2 << " " << a3;
}
// operator<< ( operator<< ( operator<< (
// operator<< ( operator<< (cout, a1), " "), a2), " "), a3)
}
```

The operator << must return a reference to the ostream to chain the calls

# Overloading >>

## Overloading global operator>> for input of user defined type

```
#include <iostream>
class A {
private:
    int a;
public:
    A (int i) : a(i) {}
    friend istream& operator>> (istream& is, A&);
};
inline istream& operator>> (istream& is, A& ra) {
    int i;
    is >> i;
    ra.a = i;
    return is;
}
int main () {
    A a (12);
    cin >> a;    // operator>> (cin, a)
}
```

# Streams Manipulators

A manipulators acts on the stream itself :

**endl** insert a new line and flush

**ends** is the same for strstreams

**flush** puts out all pending characters stored in the internal stream buffer

**oct**, **hex**, **dec** change the number base to resp. octal, hexadecimal, decimal

**ws** eats space

```
#include <iostream>
int main () {
    int i = 16;
    cout << dec << i << " " << hex << i << " " << oct << i << endl;}
16 10 20
```

# The Line Oriented Inputs

```
char buffer [10];  
cin >> buffer;           // not safe it might overflow
```

You can get a memory crash: a more secure approach is to read the input with the functions **get** and **getline**

```
char buffer [10];  
cin.get(buffer, 10, '\n');
```



# Manipulating File With Iostreams ifstream

To open a file: create an object and the constructor does the work

Do not close a file: the destructor does it for you

Call **close** explicitly if you want

```
#include <iostream>
#include <fstream>

void foo () {
    ifstream input ("IN");
    ofstream output ("OUT");
    input.close();
}
```

# Reading Lines

Open file "IN" as input, "OUT" as output, read lines from the "IN" and write them numbered into "OUT" repeat it for "OUT"

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
    int size = 100;
    char buffer [size];
    int lines = 0;
    ifstream input ("IN");
    ofstream output ("OUT");
    if (input != NULL && output != NULL) {
        while (input.getline(buffer, size)) {
            output << ++lines << ": " << buffer << "\n";
        }
    }
}

// un
// deux
// trois
// quatre
// etc ...
// 1: un
// 2: deux
// 3: trois
// 4: quatre
// 5: etc ...
```

# Open Modes

Control the way a file is opened by changing the default argument (refer to the library functions when needed)

**ios::in** read

**ios::out** write

**ios::app** append

**ios::ate** (at end) open and seek to end of file

**ios::trunc** opens a file and deletes the old file if it already exists

**ios::nocreate** opens a file only if it exists otherwise fails

**ios::noreplace** open a file only if it does not exist otherwise fails

# Open a File as Input and Output

```
#include <iostream>
#include <fstream>
int main () {
    int size = 100;
    char buffer [size];
    int lines = 0;

    ifstream input ("IN");
    // ofstream output ("IN");
    // IN will be empty to avoid it use
    ofstream output ("IN", ios::in);
    if (input != NULL && output != NULL) {
        while (input.getline(buffer, size)) {
            // IN is not modified
            output << ++lines << ": " << buffer << "\n";
        }
    }
}
```

Add the line number folowed by ": " before each line

```
#include <iostream>
#include <fstream>

int main () {
    int size = 100;
    char buffer [size];
    int lines = 0;

    ifstream input ("IN", ios::ate);
    ofstream output ("OUT");
    if (input != NULL && output != NULL) {
        while (input.getline(buffer, size)) {
            output << ++lines << ": " << buffer << "\n";
        }
    }
}
```

# Interface

When using `iostream`, you don't care where the bytes are produced or consumed

The part of the `iostream` that produces and consumes bytes is abstracted into the class **`streambuf`**

The `iostream` member function **`rdbuf`** returns a pointer to the object `streambuf`, and you can call `streambuf` member functions

Copy an `iostream` content into another `iostream` by connecting their `streambufs` using `<<`

## Send the opened file content on standard output:

```
#include <fstream>
int main () {
    ifstream input ("IN");
    if (input != null) {
        cout << input.rdbuf();
    }
}
```

## Copy from input to output:

```
#include <fstream>
int main () {
    ifstream input ("IN");
    ofstream output ("OUT");
    output << input.rdbuf();
}
```

## get with a streambuf

A form of **get** allows you to write directly into the **streambuf** of another object, the first argument is a reference to the destination streambuf, the second argument is the terminating character (which stops the get function)

```
#include <fstream>
int main () {
    ifstream input ("IN");
    while (input.get(*cout.rdbuf())) {
        input.get();
        cout << endl;
    }
}
```



## 37. The Standard Library

# The Standard Library Roles

The standard library has three major roles :

1 **portability** : supplies nonprimitive facilities that a programmer can rely on for portability, such as `list`, `vector`, `sort` functions, `I/O streams`

⇒ alternatives to hand-coded functions, classes

⇒ that stay platform—independent

⇒ essential for communication between separately developed libraries

```
#include <iostream>
#include <list> // list class library
using namespace std;

int main () {
    list<int> l;
    l.push_back (3);
    l.push_back (7);
    l.push_front (10);

    list<int>::const_iterator iter;
    for (iter=l.begin(); iter != l.end(); iter++) {
        cout << (*iter) << " ";
    }
    cout << endl;

    l.sort();

    while (l.size() > 0) {
        int value = l.front();
        cout << value << endl;
        l.pop_front();
    }
}
```

# The Standard Library Roles

2 **facilities** : provides a framework for extending the facilities it offers such as : conventions and support facilities (write the I/O of a user—defined type in the style of I/O for built—in types)

3 **extension** : provides the common foundation for other libraries

⇒ limits the scope of the standard library (you have to make choice)

⇒ places constraint on its facilities :

- list and string facilities are provided as they are often used by users in separate libraries
- graphics facilities, even if widely used, are rarely directly involved in communication between separately developed libraries

# Introduction

The standard libraries has also minor roles :

provide **language feature** such as **memory management** and **run-time type informations** (already seen)

supplies informations about **implementation—defined aspects** of the language, such as the largest float value

supplies functions that cannot be implemented optimally in the language itself for every system, such as `sqrt` and `memmove`

supplies **random—numbers** generators

...

# Standard Library Organisation

the facilities of the standard library are defined in the `std` namespace

```
using namespace std
```

they are presented as a set of headers

```
#include <...>
```

for every header **toto** part of the **C** standard library, there is a header **ctoto** defining the same names in the **std** namespace

```
/* in a c file */  
#include <toto.h>  
// in a C++ file  
#include <ctoto>
```

# Containers

The standard Library provides containers i.e. an object that contains other objects (sequence of elements)

<code>&lt;vector&gt;</code>	one-dimensional array of T
<code>&lt;list&gt;</code>	doubly-linked list of T
<code>&lt;deque&gt;</code>	double-ended queue of T
<code>&lt;queue&gt;</code>	queue of T
<code>&lt;stack&gt;</code>	stack of T
<code>&lt;map&gt;</code>	associative array of T
<code>&lt;set&gt;</code>	set of T
<code>&lt;bitset&gt;</code>	array of booleans

# Example

```
class A {  
    friend ostream& operator<< (ostream& os, A);  
    int a;  
public:  
    A(int i) : a(i) {}  
    void operator+= (A ra) { a+=ra.a; }  
};  
ostream& operator<< (ostream& os, A a) {  
    os << a.a;  
    return os;  
}  
int main () {  
    A a1(10), a2(20);  
    list<A> l;  
    l.push_back(a1);  
    l.push_back(a2);  
    cout << sum(l) << endl;  
    // 30 }  
}
```



# Standard Containers : Iterators

a container is a sequence of elements either in the order defined by the container's iterator or in reverse

<code>begin()</code>	Points to first element
<code>end()</code>	Points to one-past-last element
<code>rbegin()</code>	Points to first element of reverse sequence
<code>rend()</code>	Points to one-past-last element of reverse sequence

# Standard Containers : Direct Element Access

Some elements can be accessed directly

<code>front()</code>	First element
<code>back()</code>	Last element
<code>[]</code>	subscripting, unchecked access (not for list)
<code>at()</code>	Subscripting, checked access (for vector and deque only)

# Standard Containers : List Operations

Containers provides list operations :

<code>insert(p,x)</code>	Add x before (the element given by) iterator p
<code>insert(p,n,x)</code>	Add n copies of x before (the element given by) iterator p
<code>insert(p, first, last)</code>	Add elements from iterator first to iterator last before iterator p
<code>erase(p)</code>	Remove element at iterator p
<code>erase(first,last)</code>	Remove elements between iterator first and iterator last
<code>clear()</code>	Erase all elements

# Standard Containers : Number-of-element Operations

Containers provides list operations :

size()	Number of elements
empty()	Is the container empty
max_size()	Size of the largest possible container
capacity()	Space allocated for vector only
reserve()	Reserve space for future expansion (vector only)
resize()	Change size of container (vector, list and dequ only)
swap()	Swap elements of two containers
get_allocator()	Get a copy of the container's allocator
==	Is the content of two containers the same ?
!=	Is the content of two containers different ?
<	Is one container lexicographically before another ?

# Standard Containers : Constructors

Containers provides list operations :

<code>container()</code>	Empty container
<code>container(n)</code>	n elements default value (not for associative)
<code>container(n,x)</code>	n copies of x (not for associative)
<code>container(first,last)</code>	Initial elements from [first:last[
<code>container(x)</code>	copy constructors
<code>container()</code>	destroy container and all its elements
<code>operator=(x)</code>	Copy assignment, elements from container x
<code>assign(n,x)</code>	Assign n copies of x (not for associative)
<code>assign(first,last)</code>	Assign from [first:last]

# iterators

`<iterator>`

iterators and iterator support

# The list Standard Container

```
#include <list>
#include <iostream>
using namespace std;
ostream& operator<< (ostream& os, list<int>& l) {
    list<int>::iterator it = l.begin();
    while (it != l.end()) {
        os << *(it++) << " ";
    }
    os << endl;
    return os;
}
bool even (int i) {
    return (i % 2);
}
```

```
int main () {
    list<int> l1;
    for (int i =0; i < 10; ++i) {
        l1.push_back(i);
        l1.push_back(i);
    }
    cout << l1;
    // 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9

    l1.unique();
    cout << l1;
    // 0 1 2 3 4 5 6 7 8 9

    list<int> l2 = l1;
    cout << "equal ? " << (l1 == l2) << endl;
    // equal ? 1
}
```



```
l1.reverse();
cout << "equal ? " << (l1 == l2) << endl;
// equal ? 0

cout << "< ? " << (l1 < l2) << endl;
// < ? 0
cout << "> ? " << (l1 > l2) << endl;
// > ? 1
cout << l1.empty() << endl;
// 0

list<int>::iterator it;
for (it = l1.begin(); it != l1.end(); it++) {
    (*it)+=10;
}
cout << l1;
// 19 18 17 16 15 14 13 12 11 10
```

```
l1.insert(l1.end(), l2.begin(), l2.end());
cout << l1;
// 19 18 17 16 15 14 13 12 11 10 0 1 2 3 4 5 6 7 8 9

l1.sort();
cout << l1;
// 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

l1.remove_if(&even);
cout << l1;
// 0 2 4 6 8 10 12 14 16 18

cout << "front " << l1.front() << endl;
// front 0

cout << "back " << l1.back() << endl;
// back 18
}
```

# Input/Output

<iosfwd>	forward declarations of I/O facilities
<iostream>	and operations
<ios>	iostream bases
<iostream>	stream buffers
<istream>	input stream template
<ostream>	output stream template
<iomanip>	manipulators
<sstream>	stream to/from strings
<cstdlib>	character classification functions
<fstream>	streams to/from files
<cstdio>	printf() family of I/O
<wchar>	printf()–style I/O of wide characters

# Characters

A character set is a mapping between a character (conventional symbol) and an integer value

ASCII is the American Standard Code for Information Interchange : the ascii code of 'a' is 97

```
#include <iostream>
using namespace std;
int main () {
    cout << ((int) 'a') << endl;
    // 97
}
```

# Characters

But :

- some languages cannot be written properly using ASCII only :  
Danish, French, ...
- In different sets, different characters end up with the same integer value

⇒ The C++ approach is to allow a programmer to use any character set  
the properties of a character set are defined by its **char\_traits**

# The character traits

All `char_traits` are defined in `std`

The specialization of `char_traits` for `char` is :

```
template<> struct char_traits<char> {  
  
    typedef char char_type; // type of character  
    static void assign (char_type&, const char_type&); // = for char_type  
  
    // integer representation of char  
    typedef int int_type;  
    static char_type to_char_type (const int_type&); // int to char conv  
    static int_type to_int_type (const char_type&); // char to int conv  
    static bool eq int_type (const int_type&, const int_type&); // ==  
  
    // char_type comparisons  
    static bool eq (const char_type&, const char_type&); // ==  
    static bool lt (const char_type&, const char_type&); // <
```

```
// operations on s[n] arrays
static char_type* move (char_type* s, const char_type* s2, size_t n);
// copy n characters from s2 to s using assign(s[i], s2[i])
// works correctly if s2 is in the [s, s+n[ range

static char_type* copy (char_type* s, const char_type* s2, size_t n);
// copy n characters from s2 to s using assign(s[i], s2[i]),
// does not work correctly if s2 is in the [s, s+n[ range
// faster than move

static char_type* assign (char_type* s, size_t n, char_type a);
// assign n copies of a into s using assign(s[i], a)

static int compare(const char_type* s, const char_type* s2...);
// same as strcmp
static size_t length (const char_type* s);
static const char_type* find (const char_type* s, ...const char_type&);
static int_type eof ();  };
```

# Output Streams

An ostream is a mechanism for converting values of various types into sequences of characters

The kind of characters is characterized by `char_traits`

An ostream is the specialization for a particular kind of character of a general `basic_ostream` template

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
    virtual ~basic_ostream ();
    ...
    typedef basic_ostream<char> ostream;
    typedef basic_ostream<wchar_t> wostream;
```



# basic\_ios

The `basic_ios` base class controls formatting, locale, access to buffers and defines types for notations convenience

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef typename Tr::int_type int_type;
    // type of integer value of character
    // private and undefined assignment and copy constructor
```

`ostream` and `istream` cannot be copied (use pointer)

# Standard Streams

are declared in `<iostream>`

```
ostream cout; // standard output stream of char
ostream cerr; // standard unbuffered output stream for output messages
ostream clog; //

wostream wcout; // standard output stream of wide char
wostream wcerr;
wostream wclog;
```

# Output of Built-In Types

The class `ostream` defined the `operator<<` to handle output of the built\_in types

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
    basic_ostream& operator<< (short n);
    basic_ostream& operator<< (int n);
    basic_ostream& operator<< (long n);
    basic_ostream& operator<< (float n);
    basic_ostream& operator<< (double n);
    basic_ostream& operator<< (long double n);
    basic_ostream& operator<< (bool n);
    basic_ostream& operator<< (const void* n);
    basic_ostream& put(char_type c);
    basic_ostream& write(const char_type* p, streamsize n); // p[0]..p[n]
};
template <class Tr>
    basic_ostream<char, Tr>& operator<< (basic_ostream<Ch, Tr>&, char);
// defined using put outside the class
```

# Formatting Output of Built-In Types

there are some formatting flags in `<iomanip>`

```
cout << true << " " << false << endl;  
// 1 0  
cout << boolalpha;  
cout << true << " " << false << endl;  
// true false
```

# Virtual Output Functions

`operator<<` is not virtual

```
class Base {  
public:  
    virtual ostream& put(ostream& s) const =0;  
};  
ostream& operator<< (ostream& os, const Base& r) {  
    return r.put(os); // will use the right put  
}
```

# Input Streams

```
istream& istream::operator>> (T& tvar) {  
    // skip whitespace, then read a T into tvar  
    return *this;  
}  
  
void f () {  
    char c;  
    cin >> c;  
    char v[4];  
    cin.width(4); // read n-1 character max  
    cin >> v;  
    cout << v << endl;  
}  
  
int read_ints (vector<int>& v) {  
    int i = 0;  
    while (i<v.size() && cin>>v[i]) ++i;  
    return i;  
}
```

non-int char terminates the loop and leave the character as next to read

# Stream State

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    bool good() const; // next operation might success
    bool eof () const; // end of input seen
    bool fail () const; // next operation will fail
    bool bad () const; // stream is corrupted
    iostate rdsstate () const; // get io state flags
    void clear (iostate f = goodbit);

    operator void* () const; // nonzero if !fail()
```

# Stream State

the state is `good()` when the previous operation success

applying an input operation to a stream that is not in the `good()` state is a null operation

when the state is `fail()` but not `bad()`, the stream is uncorrupted and no character have been lost

when the state is `bad()` we never know

```
class ios_base {  
public:  
    typedef implementation_defined2 iostate;  
    static const iostate badbit, eofbit, failbit, goodbit;
```



# Input of Characters

The operator `>>` is intended for formatted input  
For char we use unformatted input function

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_istream : public basic_ios<Ch, Tr> {
public:
    streamsize gcount () const; // number of char read by last get()
    int_type get () : // read one Ch (or Tr::eof())
    basic_istream& get(Ch& c); // read one Ch into c
    basic_istream& get(Ch& c, streamsize n);
    // read at most n-1 chars
    // newline is terminator
    basic_istream& get(Ch& c, streamsize n, Ch term);

    basic_istream& getline(Ch* p, streamsize n);
    basic_istream& getline(Ch* p, streamsize n, Ch term);
```

if a `get` or `getline` does not read and remove at least one character from the stream, `setstate(failbit)` is called and subsequent reads from the stream will fail

# Input of User Defined Types

```
#include <iostream>
using namespace std;
class A {
    int a;
    friend istream& operator>> (istream& is, A& a);
public:
    int geta () const { return a; }
};

istream& operator>> (istream& is, A& a) {
    is>>a.a;
    return is;
}

int main () {
    A a1;
    cin>>a1;
    cout << a1.geta() << endl;
}
```

# File Streams

```
#include <iostream>
#include <fstream>
using namespace std;

int main (int argc, char* argv[]) {
    if (argc <3) {
        cerr << argv[0] << " inputFile outputFile\n";
        exit(1);
    }
    ifstream inputFile(argv[1]);
    if (!inputFile) {
        cerr << "cannot open input file " << argv[1] << endl;
        exit(2);
    }
    ofstream outputFile(argv[2]);
    if (!outputFile) {
        cerr << "cannot open output file " << argv[2] << endl;
        exit(3);
    }
    char c;
    while (inputFile.get(c)) outputFile.put(c);
}
```

# general utilities

<utility>	operators and pairs
<functional>	function objects
<memory>	allocators for containers
<ctime>	C-style date and time

# algorithms

<code>&lt;algorithm&gt;</code>	general algorithms
<code>&lt;cstdlib&gt;</code>	bsearch and qsort()

# diagnostics

<code>&lt;exception&gt;</code>	exception class
<code>&lt;stdexcept&gt;</code>	assert macro
<code>&lt;cerno&gt;</code>	C-Style error handling

# Strings

<code>&lt;string&gt;</code>	string of T
<code>&lt;cctype&gt;</code>	character classification
<code>&lt;cwctype&gt;</code>	wide-character classification
<code>&lt;cstring&gt;</code>	C-style string functions
<code>&lt;cwcahr&gt;</code>	C-style wide-character string functions
<code>&lt;cstdlib&gt;</code>	C-style string functions

# Localizations

<locale>	represent cultural differences
<clocale>	represent cultural C—style differences

the set of the usual conventions for natural language or culture that control the appearance of numeric values, of date, of monetary units,

... on input and output

the iostream library uses implicitly a locale that a programmer can change

```
locale myloc("POSIX"); // locale for POSIX
cin.imbue(myloc); // cin will use myloc
cin.imbue(locale()); // cin will use the default locale
```



# Language Support

<limits>	numeric limits
<climits>	C-style numeric scalar-limit macros
<cfloat>	C-style numeric floating-point limit macros
<new>	dynamic memory management
<typeinfo>	run-time type identification support
<exception>	exception-handling support
<cstdint>	C library language support : size_t, ptrdiff_t, NULL (infamous)
<cstdarg>	variable-length function argument lists
<setjmp>	C-style stack unwinding using setjmp, longjmp (best avoid use exception)
<cstdlib>	program termination
<ctime>	system clock
<csignal>	C-style signal handling

# Numeric Limits <limits>

built-in numeric types are implementation-defined and not fixed by the language itself

to deal with numbers, we need to know the general properties of the built-in numeric type

```
numeric_limits<char>::min()
-128
numeric_limits<char>::max()
127
numeric_limits<float>::min()
1.17549e-38
numeric_limits<float>::max()
3.40282e+38
```

# Numerics

<code>&lt;complex&gt;</code>	complex number and operations
<code>&lt;valarray&gt;</code>	numeric vector and operations
<code>&lt;numeric&gt;</code>	generalized numeric operations
<code>&lt;cmath&gt;</code>	standard mathematical functions
<code>&lt;cstdlib&gt;</code>	C-style random numbers

# valarray

simple single-dimensional vectors of floating-point values  
well supported by high-performance machine architecture  
the vector `valarray` is designed specifically for speed of the usual  
numeric vector operations  
intended for low-level building block for high-performance computation

# valarray

```
#include <valarray>
#include <iostream>
using namespace std;
double addone (double d) { d = d + 1; return d; }
int main () {
    valarray<double> v0;
    valarray<float> v1(1000);
    valarray<int> v2(-1, 200); // 200 int with value -1
    const double vd[] = {0, 1, 2, 3, 4, 5};
    valarray<double> v3 (vd, 6);
    for (int i = 0; i < 6; ++i) cout << v3[i] << " ";
    cout << endl;
    cout << v3.size() << endl;
    cout << v3.sum() << endl;
    cout << v3.min() << endl;
    cout << v3.max() << endl;
    valarray<double> v4 = v3.shift(3);
    for (int i = 0; i < 6; ++i) cout << v4[i] << " ";
    cout << endl;
    valarray<double> v5 = v3.cshift(3);
    for (int i = 0; i < 6; ++i) cout << v5[i] << " ";
    cout << endl;
    valarray<double> v6 = v5.apply(addone);
    for (int i = 0; i < 6; ++i) cout << v6[i] << " ";
    cout << endl;
    valarray<double> v7 = v5 * v6;
    for (int i = 0; i < 6; ++i) cout << v7[i] << " ";
    cout << endl;
}
```

# Numeric Algorithms

<accumulate(>

accumulate results of operation on a sequence

<inner\_product(>

accumulate results of operation on two sequences

<partial\_sum(>

Generate sequence by operation on sequence

<adjacent\_difference(>

Generate sequence by operation on sequence

```
template <class In, class T>
T accumulate (In first, In last, T init) {
    while (first != last) init = init + *first++;
    return init;
}
```

## 38. Nested Classes

# Nested class

```
class X {  
    struct N1 { int m; }  
    N1 foo (N2);  
public:  
    struct N2 { int m; }  
};  
void f () {  
    N1 n1; N2 n2;      // error N1 and N2 not in global scope  
    X::N1 xn1;         // error X::N1 private  
}  
N1 X::foo (N2 n2)      { ... }; // error N1 not in global scope  
X::N1 X::foo (N2 n2)    { ... }; // ok  
X::N1 X::foo (X::N2 n2) { ... }; // ok but redundant
```

Nested class just minimize the number of global name in your program



# Nested class

```
// string.H                // newstring.H                // newstring.C
class String {
    class Rep {
        ...
    };
    Rep* p;
    ...
};

// newstring.H
class String {
    class Rep;
    Rep* p;
    ...
};

// newstring.C
class String::Rep {
    ...
};
```

In the first case, if you want to change the **class Rep**, you modify **string.H** and then you must recompile all files including **string.H**. In the second case you just have to recompile **newstring.C**.

## 39. at exit, abort and exit functions

## –advanced– **atexit**, **abort** and **exit** from `<cstdlib>`

the contents of `<cstdlib>` is the same as the standard library header `<stdlib.h>` with the changes listed below :

```
int atexit (void (*f) (void))
```

the **atexit** function registers the function pointed to by **f** to be called without arguments at the normal program termination

the implementation supports the registration of at least 32 functions

the **atexit** function returns zero if the registration succeeds, nonzero if it fails

## -advanced- atexit example

```
#include <iostream>
#include <cstdlib>
using namespace std;
void first_function () {
    cout << "the first function\n"; }
void second_function () {
    cout << "the second function\n"; }
void third_function () {
    cout << "the third function\n"; }

int main () {
    cout << "beginning main\n";
    int i = atexit(&first_function);
    cout << "atexit registration " << i << "\n";
    atexit(&second_function);
    atexit(&third_function);
    cout << "ending main\n";
}
```

```
beginning main
atexit registration 0
ending main
the third function
the second function
the first function
```

## –advanced– **abort** from `<cstdlib>`

the function **abort()** terminates the program :

- without executing destructors for objects of automatic or static storage duration
- without calling the functions passed to **atexit**

## -advanced- atexit example

```
#include <iostream>
#include <cstdlib>
using namespace std;
void first_function () {
    cout << "the first function\n"; }
void second_function () {
    cout << "the second function\n"; }
void third_function () {
    cout << "the third function\n"; }

int main () {
    cout << "beginning main\n";
    atexit(&first_function);
    atexit(&second_function);
    atexit(&third_function);
    cout << "ending main\n";
    abort();
}
```

```
beginning main
ending main
Aborted
```

## –advanced– **exit** from `<cstdlib>`

```
void exit(int status);
```

objects with static storage duration are destroyed

objects with automatic storage duration are **not** destroyed

**atexit()** functions are called

if control reaches the end of the **main** without encountering an **exit**, it executes a **return(0)**

## –advanced– **exit** from `<cstdlib>`

```
#include <iostream>
#include <cstdlib>
using namespace std;
void function () { cout << "the atexit function\n"; }

class A { public: A () { cout << "A::A\n"; } ~A() { cout << "A::~~A\n"; } };
A a1;
int main () {
    cout << "beginning main\n";
    atexit(&function);
    { A a2; }
    A a3;
    exit(100);
}
```

```
A::A
beginning main
A::A
A::~~A
A::A
the atexit function
A::~~A
```



## 40. auto, register, mutable storages

# auto storage

object declared without a storage-class-specifier at block scope or as a function parameter has by default **automatic storage duration**

you can specify a storage class : **auto register static extern mutable**

**auto** and **register** :

- apply to names of objects declared in a block
- apply to function parameters
- specify that the named object has **automatic storage duration**

# auto storage

the **auto** specifier is redundant and not often used

the **register** specifier is a hint to the compiler that *the object so declared will be heavily used*

the hint can be ignored and in most implementations it will be ignored if the address of the object is taken

# 41. Run Time Type Information

# New Cast Operators

There are three new cast operators:

- **static\_cast** (not related to static keyword)
- **reinterpret\_cast**
- **const\_cast** (related to const keyword)

These new cast operators represent a **classification** of the old cast's functionality

They perform any operation the old cast can do (or they will not be accepted and old versions still used)

# const\_cast and reinterpret\_cast Operators

To ensure that the "constness" is never quietly removed, **static\_cast**, **reinterpret\_cast** can't be used to "cast away const" only **const\_cast** can

**reinterpret\_cast** is for low-level, implementation-dependent non-portable conversion

# static\_cast

**static\_cast** relies on static informations and can return a pointer to an inappropriate object

```
class A {};  
class B : public A {};  
class C : public A {};  
int main () {  
    A* pa = new B();  
    B* pb1 = (B*) pa;  
    B* pb2 = static_cast<B*>(pa);  
    C* pc1 = (C*) pa;  
    C* pc2 = static_cast<C*>(pa);  
}
```

# Run Time Type Information

When you **absolutely need** that an **object reveals** its **type** you should use **RTTI**, it allows you to **inspect** the **type** of an **object** at **run-time**

The RTTI consists of different parts :

- an operator of **dynamic\_cast** which given a base-class pointer returns a valid pointer if the object is of the expected type and 0 otherwise
- an operator **typeid** for identifying the exact type of an object given a pointer to a base class
- a structure **type\_info** giving a hook for further run-time information associated with a type



# Window, ScrolledWindow and VerticalScrolledWindow

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Window {
public:
    virtual void draw () {
        cout << "Window\n";
    }
};

class ScrolledWindow : public Window {
public:
    virtual void draw () {
        cout << "I am a ScrolledWindow\n";
    }
};

class VerticalScrolledWindow : public ScrolledWindow {
public:
    virtual void draw () {
        cout << "I am a VerticalScrolledWindow\n";
    }
};
```

# The typeid

```
void WhatIs (Window* w) {  
    if (typeid(*w) == typeid(Window)) {  
        cout << "w is a Window\n";  
    }  
    if (typeid(*w) == typeid(ScrolledWindow)) {  
        cout << "w is a ScrolledWindow\n";  
    }  
    if (typeid(*w) == typeid(VerticalScrolledWindow)) {  
        cout << "w is a VerticalScrolledWindow\n";  
    }  
}  
  
int main () {  
    WhatIs(new VerticalScrolledWindow());  
}
```

```
w is a VerticalScrolledWindow
```

# The dynamic\_cast

```
void WhatIs (Window* w) {
    w->draw();
    if (Window* ww = dynamic_cast<Window*>(w)) {
        cout << "dynamic_cast<Window>\n";
    }
    if (ScrolledWindow* sw = dynamic_cast<ScrolledWindow*>(w)) {
        cout << "dynamic_cast<ScrolledWindow>\n";
    }
    if (VerticalScrolledWindow* vsw = dynamic_cast<VerticalScrolledWindow*>(w)) {
        cout << "dynamic_cast<VerticalScrolledWindow>\n";
    }
}

int main () {
    WhatIs(new ScrolledWindow());
}
```

```
I am a ScrolledWindow
dynamic_cast<Window>
dynamic_cast<ScrolledWindow>
```

## 42. Misc. to sort

# Others modifier to variables

- **static** in global and namespace scope are invisible outside the .o
- **static** within function scope mean this variable is persistent and will be initialized on the first call of the owning function only.
- **extern** in global and namespace scope turn declaration + definition to declaration only.
- **volatile** the variable is volatile, this variable is used by another processes and can change at any time. Thus compiler must read it every time he want use it.
- **register** request the compiler to keep this variable into register. Often for optimization purpose. It's some how the oposite of volatile. Usually never use it.
- **const** tell to the compiler that you don't want modify this variable later. More on this topic later.

## –advanced– **compound** types

**Compound** types can be constructed in the following ways :

- **arrays** of objects of a given type
- **functions**
- **pointers** to void or objects or functions
- **references** to objects or functions
- **classes** containing a sequence of objects of various types
- **unions** (which are classes capable of containing objects of different types at different times)
- **enumerations** (which comprise a set of named constant values)
- **pointers** to non-static class members (which identify members of a given type within objects of a given class)

# if statements

```
if (cond0) {  
    if (cond1) {  
        f1();  
    } else {  
        f();  
    }  
}  
or  
if (cond0)  
    if (cond1) {  
        f1();  
    }  
    else {  
        f();  
    }  
}
```

The **else** is associated with the nearest un-elsed if

Use blocks !

```
if cond0 {  
    if cond1  
        f1();  
    else f();  
}
```

# Functions modifiers

- `static`: in global or namespace scope, tell the compiler to use static link. This function will not be available outside the .o
- `inline`: the compiler will try to inline the function, i.e. put the code in place instead of function call. It is used for optimization and safe macro replacement.
- `void` can be used to tell that a function does not return anything.
- `[[noreturn]]` (C++11?) tell that function never return, but, for instance, use `exit()` or `abort()`.



# constant

when you **never** need to **change** the **value** of an **object** after **initialization** : define the object as being **const**

when objects or pointers are **read** but never **written**

```
const Date date = {15, 3, 2004};  
const Date* const pDate = &date;
```

when function parameters are **read** but not **written**

```
bool same (const Date * d1, const Date * d2) { ... }
```

**const** **restricts** the **way** the **object** can be **used** : it **changes** the object **type** (it does not specify how the constant is allocated)

# this

a **member** function always **knows** for **which object** it was **invoked**

**changing** the **address** of **this** in **member functions** has no meaning and is forbidden by **const**

**inside member functions this is constant**

The type of `this` inside the member functions of a user-defined type `X` is :

```
X* const this;
```

# pointer to member

```
class T {  
public:  
    int value;  
    int foo (int, int) {};  
};  
  
typedef int (T::*IPII) (int, int);  
typedef int T::*IP;
```

```
int main () {  
    T t1;  
    T* t2 = new T ();  
    IPII p = &T::foo;  
    (t1.*p)(1, 2);  
    (t2->*p)(1, 2);  
    IP pvalue = &T::value;  
    t1.*pvalue = 12;  
    t2->*pvalue = 13;  
}
```

# volatile qualifier

**Volatile** means that the **data** may **change outside** the **knowledge** of the **compiler**

The **data** may be **changed** by **another processes**

The **compiler** make **no assumptions** about the **data**

The compiler **won't optimize code** concerning the **data**: even when it knows that it has already **read** the **data** and **never** touched to the **data**, it will **read it again**

Like **const** only **volatile member functions** can access **volatile data**

```
%makefile or Makefile (the best)
%command: make all
% tab are meaningfull
CC=g++
LD=g++
RM=rm -f
INCLUDE = /usr/X11R6/include/
LIB = /usr/X11R6/lib/ \
      $(MYLIB)/mylib.a
DEBUG = -g -Wall
CCFLAGS = $(DEBUG) -I$(INCLUDE)
LDFLAGS= -L$(LIB)
HSOURCES = sort.h
CSOURCES = sort.cpp \
          main.cpp
OBJECTS = main.o sort.o
```

```
all : sort

sort.o: sort.cpp $(HSOURCES)
main.o : main.cpp $(HSOURCES)

.SUFFIXES: .cpp .h

.cpp.o: ; $(CC) -c $(CCFLAGS) $<

sort : $(OBJECTS)
      $(LD) -o $@ $(OBJECTS) $(LDFLAGS)

clean :
      $(RM) *~ *.o sort
```

## overloading << to print an IntStack

You can implement for all your classes the ostream << operator

the typical way to do it is to put the following functions as a friend of your class (for the operator to have access to the class private fields)

```
class IntStack {  
    friend std::ostream& operator<< (std::ostream& os, const IntStack& s);  
    // the rest of the class definition  
};
```

You can then use it inside a classical c++ stream output

```
#include <iostream>  
using namespace std;  
int main () {  
    IntStack s (12);  
    s.push(1);  
    s.push(4);  
    cout << s << endl;  
}
```