Software Engineering
Fundamentals of C++

Yohan FOUGEROLLE,
Département Ge2i, Bureau 005,

Centre Universitaire Condorcet
Laboratoire Le2i - 12 rue de la Fonderie - 71200 LE CREUSOT

Tel : 03 85 73 11 37 ✉ yohan.fougerolle@u-bourgogne.fr

Course Syllabus

1. Introduction
2. Basics in C++
    - First 6 – 8 hours of class
3. Oriented object programming skills
    - For all the rest of the semester.
4. Useful (life saving) libraries
    - Standard Template Library (STL)
    - Eigen library
    - Use of OpenGL and OpenCV libraries
5. Classical algorithms (sort, search, etc)
6. Advanced algorithms (PCA, LU, etc)
    - Some of these algorithms can also be studied in the Applied Mathematics module

In practice...

- Professors
  - Lectures + project follow up : Y. Fougerolle
  - Labs : 1 Ph.D Student + Y. Fougerolle
- All ressources available online (Labs, slides, cpp examples, etc)

Total number of class hours (lectures + labs) ~ 50h

Course assessment

- Course work and Labs
  - 1 **average** Course work mark (CW)
  - When do you return your assignment?
- 1 Project
  - Final mark = (1*CW+9*PM) / 10
  - Practical details available online

Quantitative course structure

1. **C++ Lectures (~20h) :**
   - **Basics (~6h)**
   - Oriented Object Programming(~4h)
   - Advanced concepts (~2h)
   - The Standard Template Library (~4h)
   - OpenGL and OpenCV Libraries (~2h)

2. **C++ Labs 10 x 2h**

3. **Project follow up and interactive courses (10h)**

C++ general overview

- Few facts
  - statically typed, free-form, multi-paradigm programming language.
  - Middle-level language (combination of both high-level and low-level language features)

- Developed by Bjarne Stroustrup starting in 1979 at Bell Labs. Originally named "C with Classes", renamed to C++ in 1983

- **VERY** widely used in the software industry : systems/application/entertainment/embedded softwares, device drivers, high-performance server and client applications, etc

- Several groups provide both free and proprietary C++ compiler software : The GNU Project, Microsoft, Intel, Borland, see references and useful links

- Standards
  - ratified in 1998 as ISO/IEC 14882:1998 (C++98)
  - amended by the 2003 technical corrigendum, ISO/IEC 14882:2003
  - C++11 (formerly known as C++0x) since 2011

References and useful links

- References
  - Stroustrup, Bjarne (2000). The C++ Programming Language
  - Cheapest, most exhaustive: google...

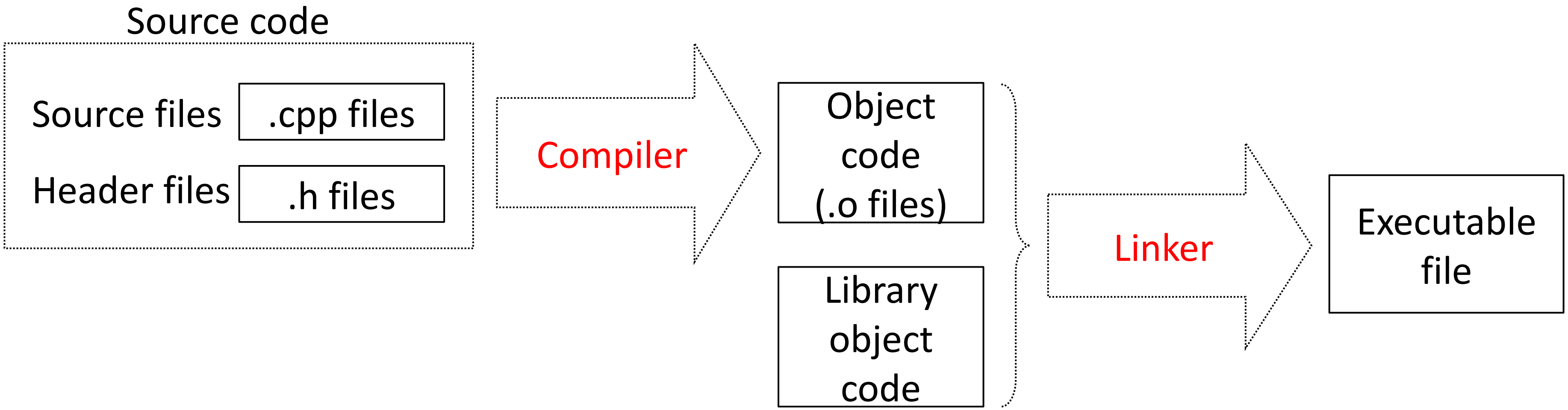
- Useful links for compilers (Windows only, non exhaustive)

  - Open source (free for non commercial applications, highly recommended)
    - Dev cpp (simple and easy, beta version sometimes unstable): http://www.bloodshed.net/
    - Code::blocks (multi compiler, still simple, efficient, debugger so so): http://www.codeblocks.org/
    - **Qt SDK** : http://qt.nokia.com/

  - Retail: free (or student) versions can be dowloaded
    - Microsoft Visual Studio http://www.microsoft.com/express/download/#webInstall
    - C++ builder : http://www.embarcadero.com/products/cbuilder

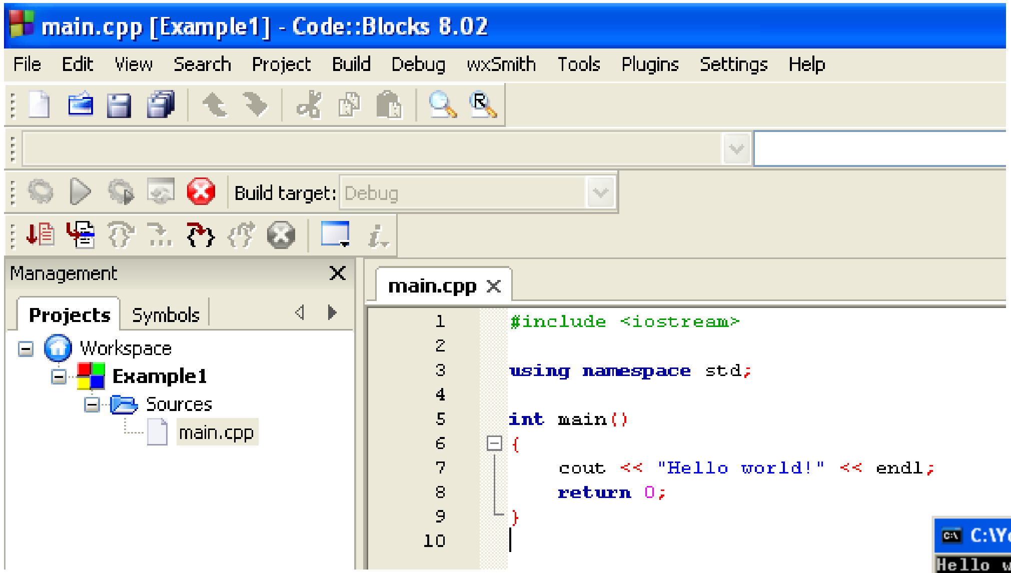
- Few advices
  - Whatever happens, for source codes, tutorials, etc: use the web!
  - Share your experience and problems with other students and faculties.

Compiling and linking



- You write C++ source code (in principle human readable), some libraries are sometimes provided as a bunch of .h, .cpp or .cxx files (Eigen for instance)
- The compiler translates what you wrote into object code (sometimes called machine code)
  - Object code is simple enough for a computer to "understand"
- The linker links your code to system code needed to execute
  - E.g. input/output libraries, operating system code, windowing code, or specific libraries (OpenCV, CGAL), etc.
- The result is an executable program

"Hellow world" example

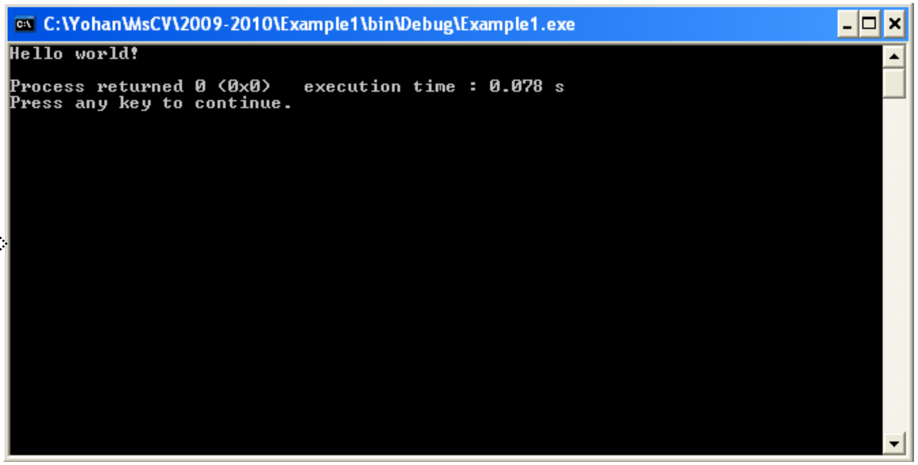

main.cpp [Example1] - Code::Blocks 8.02
File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Help
Build target: Debug
Management
Projects Symbols
Workspace
Example1
Sources
main.cpp
main.cpp

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}

```

Simplest structure

- 1 file main.cpp
- No header file

1. Console application project
2. Compile it
3. Run it



```
Hello world!

Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

"Hellow world" example

```
#include <iostream>

using namespace std;

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

- Lines beginning with a hash sign (#) are directives for the preprocessor indications for the compiler's preprocessor)
- `#include <iostream>` tells the preprocessor to include the iostream standard file.
  - This specific file (iostream) includes the declarations of the basic **standard input-output** library in C++
  - iostream is included because its functionality is going to be used later in the program (for `cout, <<`, and `endl`).
- When you will write your own projects, you will also use your own header files (for classes, interface, useful functions, etc)

"Hellow world" example

```
#include <iostream>

using namespace std;

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

- All the elements of the standard C++ library are declared within what is called a namespace. On this example, the standard namespace is named `std`

- To access and use the functions/variables/constants of a given namespace

- This line is very frequent (almost systematic) in C++ programs

- Later, you may use several namespaces or even write your own ones

Parameters in the main function??

```
int main (int argc, char ** argv)
int main (int argc, char *[] argv)
```

- The argc parameter is **the number of command line options** specified, including the executable name, when the executable was invoked.
- The individual command line options are found in the argv array, which is NULL terminated (the name and path used to invoke the executable is in argv[0]).
- Example, you might run your .exe file and pass some filenames to read the data and write the output by typing something similar to:

```
myprogram.exe inputdata.dat outputdata.dat
```

"Hellow world" example

```
#include <iostream>

using namespace std;

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

- This line is a C++ statement: simple or compound expression

- cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case "Hello World") into the standard output stream (which usually is the screen).

- cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file earlier in our code.

- Notice that the statement ends with a semicolon character **;**
  - This character is used to mark the end of the statement
  - Must be included at the end of all expression statements in all C++ programs

"Hellow world" example

```
#include <iostream>

using namespace std;

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

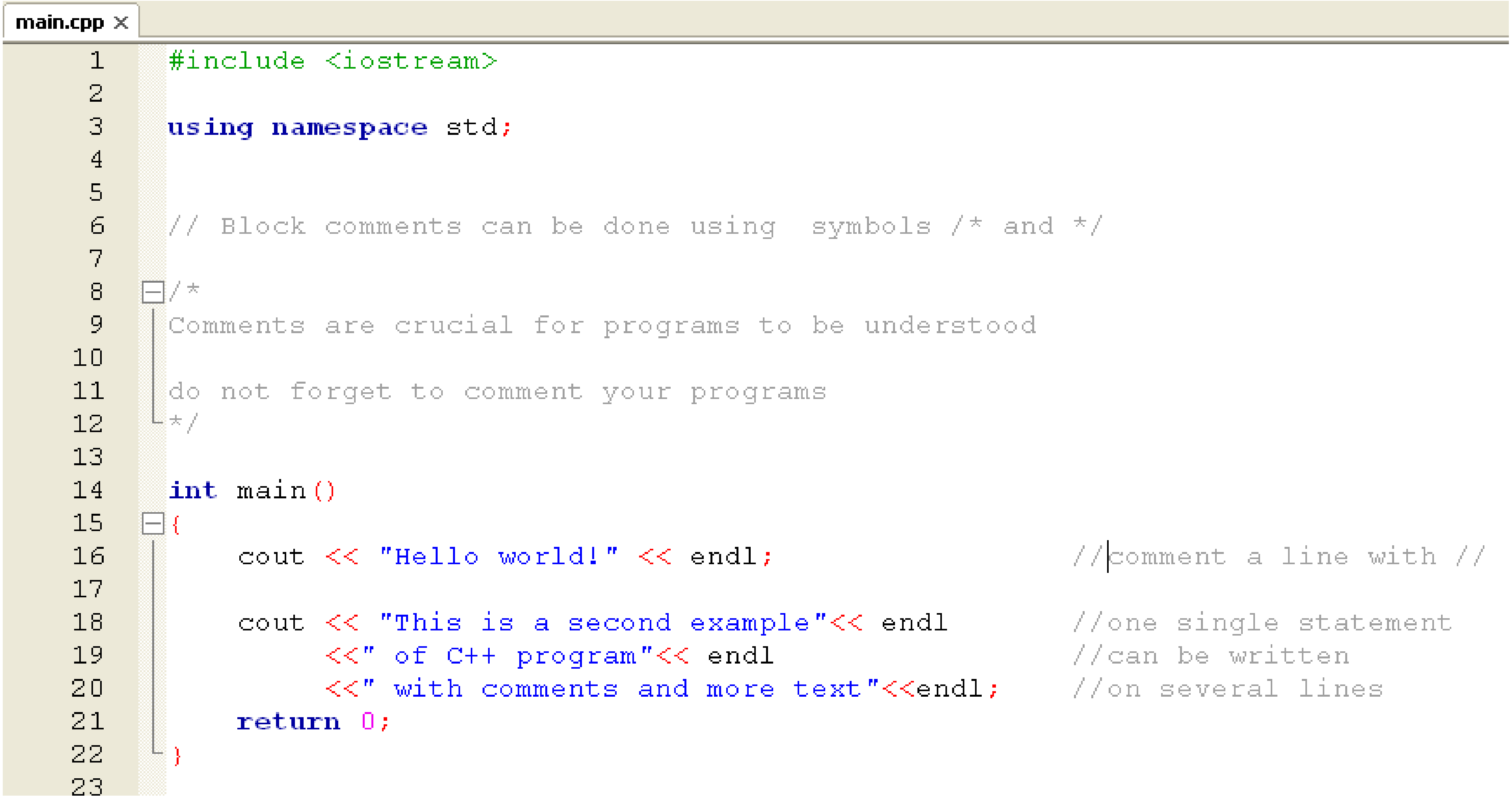
- The return statement causes the main function to finish

- Return may be followed by a return code (in our example, 0)

- A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution

- This is the most usual way to end a C++ console program
- You may use your own return values, or standard constants

Example 2



```cpp
#include <iostream>

using namespace std;


// Block comments can be done using  symbols /* and */

/*
Comments are crucial for programs to be understood

do not forget to comment your programs
*/

int main()
{
    cout << "Hello world!" << endl;                 //comment a line with //

    cout << "This is a second example"<< endl       //one single statement
         <<" of C++ program"<< endl                 //can be written
         <<" with comments and more text"<<endl;    //on several lines
    return 0;
}

```

- Line comments with //
- Block comments with /* and */
- Comments are very important to understand a source code

Identifiers

- A valid identifier is a sequence of one or more letters, digits or underscore characters
- Neither spaces nor punctuation marks or symbols.
- Variable identifiers always have to begin with a letter
- They can also begin with an underline character _, but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere.

In no case they can begin with a digit

- They cannot match any keyword of the C++ language nor your compiler's specific ones, which are reserved keywords. The standard reserved keywords are:

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

Careful: C++ is a case sensitive language

Fundamental data types

Name	Description	Size	Range
char	Character or small integer	1 byte	signed: -127 to 127 unsigned : 0 to 255
short int (short)	Short integer	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer	8 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
float	Floating point number.	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number	8 bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character	2 or 4 bytes	1 wide character
bool	Boolean value. It can take one of two values: true or false.	1 byte	true or false

Declaration of variables

In order to use a variable in C++, you must declare it

Syntax

```
<Type> <Name>;
```

For example:

```
int a;    //declares a variable of type int with the identifier a
float mynumber; // same for a float identified by mynumber
```

Once declared, the variables a and `mynumber` can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, separate their identifiers with commas:

```
int a, b, c; //declare 3 variables of type int called a, b, and c
```

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented.

Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero).

Can be specified by using either the specifier signed or the specifier unsigned before the type name. For example:

```
unsigned short int NumberOfSisters; // >=0 whatever happens
signed int MyAccountBalance; // can be < 0 !
```

Scope of variables

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

int gVar1, gVar2;

int main ( int argc, char**argv) //notice the arguments here
{

     int num = 5;
     cin >> gVar1 >> gVar2;

     cout << gVar1-num << "   " << gVar2 + num << endl;

     return 0;
}
```

2 global variables

1 local variable

- Global variables can be **referred from anywhere in the code**, even inside functions, whenever it is after its declaration. **Avoid global variable as much as possible**. To access global variables declared outside of a file where you need them, use the keyword **`extern`**

- The scope of local variables is limited to the block enclosed in braces **{}** where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function.

Initialisation

When declaring a variable, its value is by default **undetermined**

- If it is statically allocated, it is initialized at the default value
- If it is dynamically allocated, it is not initialised

There exist two solutions to initialise a variable

1. Append an equal sign followed by the value to which the variable will be initialized:

```
<type> <identifier> = <initial_value>;
```

Example: `int a = 5;`

2. Initialize variables (constructor initialization), by enclosing the initial value between parentheses ()

```
<type> <identifier> ( <initial_value> ) ;
```

Example: `int a(5);`

Both ways of initializing variables are valid and equivalent in C++, constructor initialisation is slightly faster (supposedly).

Assignment =

- The assignment operator assigns a value to a variable.

  ```
  a = 12; // store the value 12 into the variable a
  ```

- The part at the left of the assignment operator = is known as the lvalue (left value)
- The part at the right of the assignment operator = one is known as the rvalue (right value).
  - lvalue has to be a variable
  - rvalue can be either a constant, a variable, the result of an operation or any combination of these.

- The most important rule when assigning is the right-to-left rule: The assignment operation always takes place from **right to left**
  - `a = b;`
  - This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost.

- Assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation:

```
a  =  3 * ( b = 2 ); // avoid such things
```

Is equivalent to `b = 2; a = 3 * b;`

```
a = b = c = 4; // 4 is assigned to variables a, b, and c... in which order??
```

Arithmetic operators

+ : addition

- : substraction

/ : division

* : multiplication

  Warning, this operator can also have a complete different meaning, see pointers in few lectures

  Usually the arithmetic operator and related pointer operator are easy to distinguish

% : modulo

  gives the remainder of a division of two values.

  For example:

```
A = 10 % 7;
// A will be set to the remainder of
// the integer division of 10 by 7, i.e. 3
```

Compound assignments

- Arithmetic operators (and bitwise operators, see further) can be combined with the affectation operator =

<table>
<tr><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>&lt;&lt;=</td><td>&gt;&gt;=</td><td>&amp;=</td><td>^=</td><td>|=</td></tr>
</table>

Examples

```
-   a = a + 2;   // is equivalent to      a += 2;
-   b = b * .33; // is equivalent to      b *= .33;
-   c = c / 4;   // is equivalent to      c /= 4;
-   d = d * 10;  // is equivalent to      d *= 10;
```

- Be VERY careful with type conversions...

```
int a(2), b(3), c;
c = a/b; // What is the value of c?
```

Increase ++ and decrease --

- `a++;` is equivalent to `a = a + 1;` and also to `a += 1;`
- `b--;` is equivalent to `b = b - 1;` and also to `b -= 1;`

- Characteristic: can be **used both as a prefix and as a suffix**.
  - ➔ it can be written either before the variable identifier (++a) or after it (a++)
  - ➔ Be EXTREMELY careful with precedence of operators…
- In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression
- In case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression

`a = 2;`
`b = a++;`
	`a = 2;`
	`b = ++a;`

Here a=3 and b=2 (a incremented after affectation)	Here a=3 and b=3 (a is incremented before the affectation)

Relational and equality operators

- In order to evaluate a comparison between two expressions we can use the relational and equality operators

- The result of a relational operation is a **boolean value**
  - ➔ Can only be **true or false**

- List of the relational and equality operators that can be used in C++

==	Equal to	5 == 2 // will return false
!=	Not equal to	3 != 7 // will return true
>	Greater than	8 > 2 // will return true
>=	Greater than or equal to	5 >= 5 // will return true
<	Less than	5 < 5 // will return false
<=	Less than ot equal to	5 <= 4// will return false

- Be careful, do not misunderstand operators = (assignment) and == (comparison)

Logical operators !, &&, and ||

- The Operator ! is the C++ operator to perform the boolean operation NOT
  - Only one operand, located at its right
  - It returns the opposite Boolean value of evaluating its operand

```
! (5 == 5)    // false because the expr. at its right (5 == 5) is true
! (6 <= 4)    // true because (6 <= 4) would be false.
! true        // evaluates to false
! false       // evaluates to true.
```

- The logical operators && and || are used when evaluating two expressions to obtain a single relational result
- The operator && corresponds to the boolean logical operation AND.
- The operator || corresponds to the boolean logical operation OR.

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

Conditional operator ? :

- Evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false

- Syntax

```
        <condition> ? <result1> : <result2>
```

- Examples

```
7 == 5 ? 4 : 3         // returns 3, since 7 is not equal to 5
7 == 5 + 2 ? 4 : 3     // returns 4, since 7 is equal to 5+2
5 > 3 ? a : b          // returns the value of a
a > b ? a : b          // returns whichever is greater, a or b
```

- Can be used to simplify elementary if / else / return structures

Comma operator ,

- Used to separate two or more expressions that are included where only one expression is expected

- When the set of expressions has to be evaluated for a value, only the **rightmost** expression is considered

- For example

```
a = ( b = 3, b + 2 ); // once again, avoid this
```

  - Would first assign the value 3 to b
  - Then assigns b+2 to variable a.
  - At the end, variable a contains the value 5 while variable b contains value 3

- This is equivalent to

```
b = 3;
a = b + 2;
```

- Suggestion: avoid such cumbersome syntax, maybe confusing (see numerical recipies in C/C++ for various examples)

Bitwise operators

- Bitwise operators modify variables considering the bit patterns that represent the values they store

Operator	ASM equivalent	Description
&	AND	Bitwise And
|	OR	Bitwise Or
^	XOR	Bitwise Xor
~	NOT	Unary Complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

- Careful to the confusion between boolean/relational operators and bitwise operators...
- Operators << and >> can be used with streams (screen, files, other)

Other operators

- Explicit type casting
  - To convert a type to another type
  - The simplest one is to precede the expression to be converted by the new type enclosed between parentheses ()
  - The second one is to use the type itself as a function.

```
int i;
float f = 3.14;
i = (int) f;                 // old C style
i = int ( f );               // functional notation
```

  - For more information on efficient techniques to cast, search for `static_cast, reinterpret_cast, etc`

- Sizeof ( )
  - Returns the size in bytes of a type or an object

```
a = sizeof (char);
```

  - This will assign the value 1 to a because char is a one-byte long type

- Pointer operators, * , & , ... later
- Beware of unadapted casts, truncations, etc

Precedence of operators

- When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later.

- Ex: does the expression a = 5 + 7 % 2 mean

```
➔   a = 5 + (7 % 2)    // with a result of 6, or
➔   a = (5 + 7) % 2    // with a result of 0
```

- Correct answer is the first of the two expressions, with a result of 6.

- Operators have different priorities, see table next slide

- To avoid confusion use parentheses ( ) as often as possible

Precedence of operators

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	
	* &	indirection and reference	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= |=	assignment	Right-to-left
18	,	comma	Left-to-right

Control Structures

- With the introduction of control structures we are going to have to introduce a new concept: the compound-statement or block.
- A block is a group of statements, separated by semicolons (;) grouped together in a block enclosed in braces:

```
{
    statement1;
    statement2;
    statement3;
}
```

- A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block)

- In the case that we want the statement to be a simple statement, we do not need to enclose it in braces { }

- A compound statement **MUST** be enclosed between braces { } to form a block
- Forgetting { } is a common source of error, as a recommendation, always use { }

Conditional structure: if and else

- The if keyword is used to execute a statement or block only if a condition is fulfilled

- Syntax

```
if ( <condition> ) <statement>;
or
if ( <condition> ) { <block of statement>; }
```

- If condition is true, statement is executed
- If condition is false, statement is ignored
  - → not executed
  - → the program continues right after the conditional structure

- Example

```
if (x == 100)    cout << "x is 100";
```

- If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
    cout << "x is ";
    cout << x <<endl;
}
```

Conditional structure: if and else

- Can additionally specify if the condition is not fulfilled by using the keyword `else`
- Its form used in conjunction with if is:

```
if ( <condition> ) { <statement(s)>; } else { <statement(s)>; }
```

- For example:

```
if (x == 100)        cout << "x is 100";
else                 cout << "x is not 100";
```

- This codes prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100

- Several if + else structures can be concatenated with the intention of verifying a range of values :

```
if (x > 0)              cout << "x is positive";
else if (x < 0)         cout << "x is negative";
else                    cout << "x is 0";
```

- Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

The selective structure switch

- To check several possible constant values for an expression

- Syntax

```
switch ( <expression> )
{
    case <constant1> :
        {<group of statements 1;>}
        break;

    case <constant2> :
        {<group of statements 2;>}
        break;


       .
       .
       .
    default:
        {default group of statements;}
}
```

The selective structure: switch

- `switch` evaluates expression and checks if it is equivalent to `constant1`
  - if yes
    - executes group of statements 1 until it finds the break statement.
  - If no, then checks against constant2.
    - If yes, executes group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure
    - Etc

- Finally, if the value of expression did not match any of the previously specified constants
  - Executes the statements included after the `default:` label, if it exists (since it is optional)
  - Never forget the `default:` case, it is optional
  - Good programmers always need it (error recovery, debugging, etc)

The selective structure switch

<table>
<tr><th>switch example</th><th>if-else equivalent</th></tr>
<tr><td><pre>
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
}
</pre></td><td><pre>
if (x == 1) {
  cout << "x is 1";
  }
else if (x == 2) {
  cout << "x is 2";
  }
else {
  cout << "value of x unknown";
  }
</pre></td></tr>
</table>

- The switch statement uses labels instead of blocks
  - Forces to put break statements after the group of statements
  - Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Iteration structures (loops)

- Loops repeat a statement
  - ➔ certain number of times (for)
  - ➔ while a condition is fulfilled
- while loop

  `while ( <expression> ) { <statement(s);> }`
- Repeats statement **while** the condition set in expression is **true**.

Example

<table>
<tr><th>Main program</th><th>Result</th></tr>
<tr><td><pre>
#include <iostream>
using namespace std;
int main (){
  int n;
  cout << "Enter the starting number > ";
  cin >> n;
  while (n>0) {
    cout << n << ", ";
    n = n-1;
  }
  cout << "FIRE!\n";
  return 0;
}
</pre></td><td><pre>
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
</pre></td></tr>
</table>

Iteration structures (loops)

Step by step analysis

1. User assigns a value to n

2. The while condition, n>0, is checked
   condition is true: statement is executed (to step 3)
   condition is false: ignore statement and continue after it (to step 5)

3. Execute statement

```
cout << n << ", ";       // prints the value of n
n = n - 1;               // decreases n by 1
```

4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program

Reminder :

- When using a while-loop, **always** provide (within the block) some method to force the condition to become false at some point, otherwise the loop will continue looping forever.
  - In this case we have included `n = n - 1;`
  - decreases the value of the variable that is being evaluated in the condition (n)
  - eventually the condition (n>0) becomes false after a certain number of loop iterations

The do-while loop

- Syntax

  ```
  do { <statement(s)> } while ( <condition> );
  ```

- Same as the while loop, except that condition in the do-while loop is evaluated **after** the execution of statement instead of before
- At least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

Main program

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

int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

Result

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The for loop

- Syntax

```
for ( <initialization>; <condition>; <increase> )
{
    <statement(s)> ;
}
```

- Its main function is to repeat statement while condition remains true, like the while loop

- Designed to perform a repetitive action with a counter which is initialized and increased at each iteration

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

The for loop

Main program	Result

// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
	10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
	FIRE!

- The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written

```
for (;n<10;)      //No initialization and no increase
for (;n<10;n++)   //increase field but no initialization
```

- Using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop

```
for ( n=1, i=100 ; n!=i ; n++, i-- )          {  // whatever here... }
          /* When does this loop stop?? */
```

Number of iterations?

```
for (int i = 0; i < 10; i++)      // 9, 10 or 11 iterations??
{
   /* do something here*/
}

for (int i = 10; i > 0; i--)    //same here
{
   /* do something here*/
}

for (int i = 0; i % 10 != 0; i+= 3 ) // what does this mean?
{
   /* do something here*/
}

for (int i = 0; i < 10; i++ )        // careful with indices
  for (int j = 0; j < 10; j++ )      // index confusion is a very common source of errors
  {
   /* do something here*/
  }

for (int i = 0; i < 10; i++ )        // careful with indices
  for (int j = 0; j <= i; j++ )      // index confusion is a very common source of errors
  {
   /* do something here*/
  }
```

Jump statements

- The `break` statement
  - → Already seen with `switch` structure
  - → Used to leave a loop even if the condition for its end is not fulfilled
  - → Can be used to end an infinite loop, or to force it to end before its natural end

Main program

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << endl << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

Result

```
10, 9, 8, 7, 6, 5, 4, 3,
countdown aborted!
```

The continue statement

- The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.

- Example:

Main program

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

Result

```
10, 9, 8, 7, 6, 4, 3, 2, 1,
FIRE!
```

Functions

- Using functions we can **structure our programs** in a more modular way.
- A function is a group of statements that is executed when it is called from some point of the program.
- Syntax

```
<type> <name> ( <param1>, <param2>,...) { <statements;> }
```

where:

- `<type>` is the **data type** specifier of the **data returned by the function**.
- `<name>` is the identifier by which it is possible to **call the function**.
- `<parameters>` (as many as needed): data type specifier followed by an identifier which **acts within the function as a regular local variable**.
  - allow to pass arguments to the function when it is called.
  - Different parameters are separated by commas.
- `<statements;>` is the function's body surrounded by braces { }.

Functions: A first (and dirty) example

- Here, the function is declared and implemented within the main.cpp file...

Main program

```cpp
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;     //local variable
  r = a + b;

  // return the value of r
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

Result

```
The result is 8
```

Note on declaring functions

- Until now, we have defined all of the functions before the first appearance of calls to them in the source code (generally in function `main` which we have always left at the end of the source code).

- If you place the function `main` before any of the other functions that were called from within it, you will obtain compiling errors.
  - To call a function **it must have been declared** in some earlier point of the code
  - There is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function:
    - Just declare prototype / signature of the function before it is used, instead of the entire definition (usually in the header files).
    - This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

- Syntax

```
<type> <name> ( argument_type1, argument_type2, ...);
```

- It is identical to a function definition, except
  - It does not include the body of the function itself
  - No need for variable names, only types matter
- We end the prototype declaration with a mandatory semicolon ;

Main program

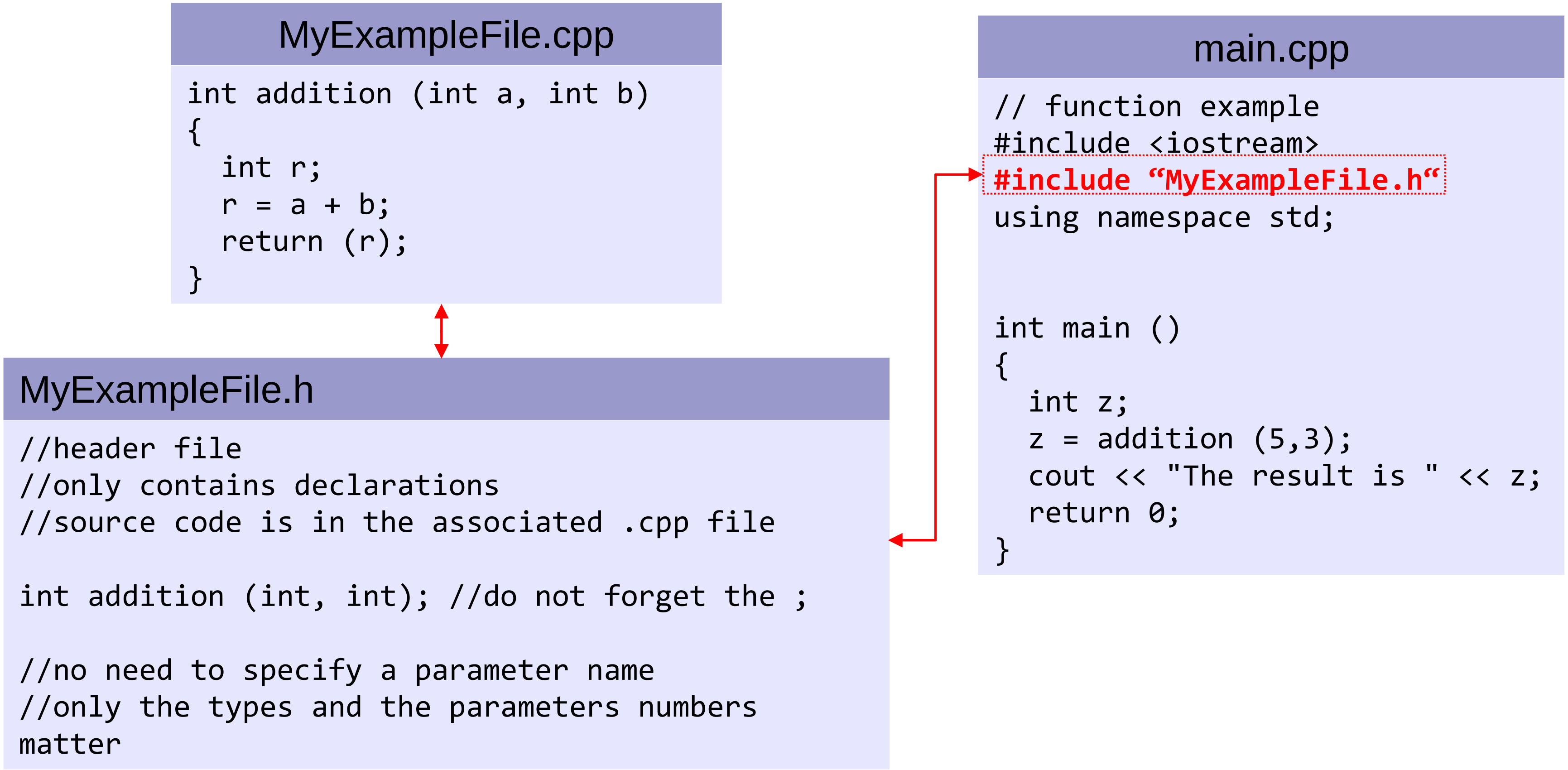
```
// declaring functions prototypes
// still dirty: prototypes are declared in the main.cpp file
// rather declare function prototypes in header file

#include <iostream>
using namespace std;

void odd (int);  // function prototype

int main (){
  int i;
  do {
    cout << "Type a number (0 to exit): "; cin >> i;
    odd (i);
  }
  while (i!=0);

  return 0;
}

//now comes the implemtation of function odd
//notice that the body can be implemented after its call

void odd (int a){

  if ( a%2 != 0 )     cout << "Number is odd.\n" << endl;
  else                cout << "Number is even.\n" << endl;
}
```

Result

```
Type a number (0 to
exit): 9
Number is odd.
Type a number (0 to
exit): 6
Number is even.
Type a number (0 to
exit): 1030
Number is even.
Type a number (0 to
exit): 0
Number is even.
```

Functions: A better example

- Separate the declaration and the implementation of the function
  - Declaration in header file (.h)
  - Implementation in source file (.cpp)
  - Include your header file to use the function



MyExampleFile.cpp

```
int addition (int a, int b)
{
  int r;
  r = a + b;
  return (r);
}
```

MyExampleFile.h

```
//header file
//only contains declarations
//source code is in the associated .cpp file

int addition (int, int); //do not forget the ;

//no need to specify a parameter name
//only the types and the parameters numbers
matter
```

main.cpp

```
// function example
#include <iostream>
#include “MyExampleFile.h“
using namespace std;


int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

The `void` type

- `void` specifies that we want the function to take no parameters when it is called:

```
void printmessage (void)
{
  cout << "I'm a function"<<endl
    << "without parameters"<<endl
    << "and without returned data"<<endl;
}
```

- It is optional (but preferable) to specify `void` in the parameters list.

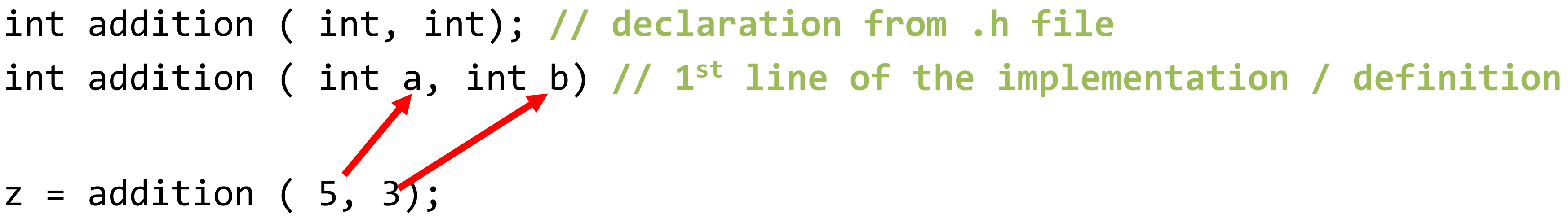
- The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to printmessage is:

```
    printmessage(); // correct call
    printmessage;   // incorrect call
```

- The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement.

Function parameters

- The main function begins by declaring the variable z of type int
- Then, we see a call to a function called `addition(int, int)`.
- See the logical link between the structure of the call to the function and the declaration of the function:



```
int addition ( int, int); // declaration from .h file
int addition ( int a, int b) // 1st line of the implementation / definition

z = addition ( 5, 3);
```

- At the point at which the function is called, the control is lost from the `main` function and passed to the function `addition`.
- The value of both arguments passed in the call (5 and 3) are **copied** to the **local variables** int a and int b.
- Function addition
  - declares another local variable (int r)
  - assigns to r the result of a+b
  - Since the values passed for a and b are 5 and 3, the function returns the value 8

Returned data

- The following line of code:

```
return (r);
```

  - Finalizes function `addition`
  - Returns the control back to the function that called it in the first place (in this case, main).
  - The program follows it regular course from the same point at which it was interrupted by the call to `addition`.

- The return statement **specifies a value** which is the content of variable r
  - This value becomes the value of evaluating the function call:

```
int addition ( int a, int b)
 8 ↓
z = addition ( 5, 3);
```

- The variable z is set to the value returned by addition (5, 3), that is 8.
- In other words, you can imagine that the call to the function `addition (5,3)` is literally replaced by the value it returns, i.e. 8.

Examples

Main program

```cpp
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r = a - b;
  return (r);
}

int main ()
{
  int x=5, y=3, z;

  // call by providing values
  // and storing results into a local variable z

  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';

  // Same without storage, result directly used through cout <<....
  cout << "The second result is " << subtraction (7,2) << '\n';

  //call using variables as parameters, result directly used as well
  cout << "The third result is " << subtraction (x,y) << '\n';

  return 0;
}
```

Result

```
The first result is 5
The second result is 5
The third result is 2
```

Functions: using parameters

- Parameters / Arguments passed by value and by reference.
  - Until now, in all the functions we have seen, the arguments passed to the functions have been passed by value:

We have passed to the functions local copies but never the variables themselves.

  - For example, suppose that we called our first function addition:

```
int x = 5, y = 3, z;
z = addition ( x , y );
```

Here, we call the function `addition` passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves. This way, when the function is called, the value of its parameters a and b become 5 and 3, respectively.

Any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

- There might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose pass arguments by reference using the symbol **&**.

Example

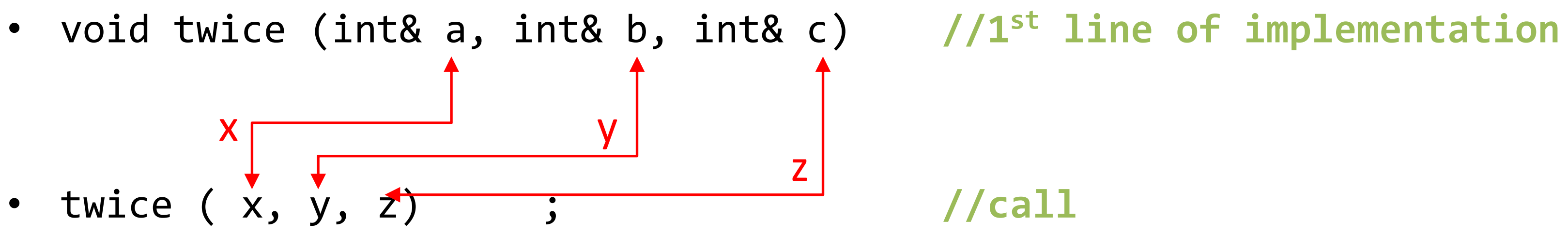
Main program

```
// passing parameters by reference

#include <iostream>
using namespace std;

void twice (int& a, int& b, int& c)
{
  a *= 2;
  b *= 2;
  c *= 2;
}

int main ()
{
  int x=1, y=3, z=7;
  cout << "x=" << x << ", y=" << y << ", z=" << z << endl;

  //call the function
  twice (x, y, z);

  // now display the result
  cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
  return 0;
}
```

Result

```
x=1, y=3, z=7
x=2, y=6, z=14
```

Functions



```
• void twice (int& a, int& b, int& c)     //1st line of implementation

• twice ( x, y, z)      ;                  //call
```

- Each parameter type is followed by a symbol ampersand &
  - Specifies that the corresponding arguments are to be passed by reference instead of by value
  - When a variable is passed by reference we are not passing a copy of its value, but the variable itself to the function
  - Any modification to the local variables will have an effect in their counterpart variables passed as arguments

- To explain it in another way
  - We associate a, b and c with the arguments passed on the function call (x, y, and z)
  - Any change on variable a within the function will affect the value of x outside it (same for b and y, and c and z)

Functions

Passing by reference allows a function to return **more than one value**.

<table>
<tr><th>Main program</th><th>Result</th></tr>
<tr><td><pre>
// more than one returning value
#include <iostream>
using namespace std;

void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}

int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
</pre></td><td><pre>
Previous=99, Next=101
</pre></td></tr>
</table>

Default values in parameters

- When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

- Use the assignment operator = and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used

Main program

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r =a / b;
  return (r);
}

int main (){
  cout << divide (12); // 2nd parameter not provided
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

Result

```
6
5
```

Overloaded functions

- You can give the same name to more than one function if they have either a different number of parameters or different types in their parameters
- The returned type is not sufficient to overload a function (guess why)

Main program

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

//integer parameters and return type
int operate (int a, int b)       {
  return (a * b);
}

//float parameters and return type
float operate (float a, float b) {
  return (a / b);
}

// overloaded function (same name, but different parameter types)

int main (){

  int x=5, y=2;
  float n=5.0, m=2.0;
  cout << operate (x,y) << "\n"; // integer params-->first function
  cout << operate (n,m) << "\n"; // float params → second function
  return 0;
}
```

Result

```
10
2.5
```

Overloaded functions

- We have defined two functions with the same name, `operate`, but one of them accepts two parameters of type `int` and the other one accepts them of type `float`:

```
//integer parameters and return type
int operate (int a, int b)
{ return ( a * b ); }
```

```
//float parameters and return type
float operate (float a, float b)
{return ( a / b );}
```

  - The compiler knows which one to call in each case by examining the types passed as arguments when the function is called:
    - In the first call to operate, the two arguments passed are of type int, therefore, the function with the first prototype is called
    - The second call passes two arguments of type float, so the function with the second prototype is called

- A function **cannot be overloaded only by its return type**
  - In such case, the compiler cannot determine which function to use
  - As a consequence, at least one of its parameters must have a different type

Inline functions

- The `inline` specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function.
  - Does not change the behavior of the function itself
    - Suggests to the compiler that the code generated by the function body is inserted at each point the function is called
    - Generally avoids some additional overhead in running time
- The format for its declaration is:

```
inline type name ( arguments ... ) { instructions ... }
```

- For instance

```
    inline double sqr(double x) {return x*x;}
```

- The call is just like the call to any other function
  - You do not have to include the inline keyword when calling the function, only in its declaration
  - Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Inline function vs macros

- Macros are unsafe (and evil) : no verification of the parameters (type, instruction, etc). Can be seen as a textual copy and paste…

```
// a (evil) macro which returns an absolute value
#define unsafe(i) ( (i) >= 0 ? (i) : -(i) )

// same idea using (good) inline function
inline int safe(int i) { return i >= 0 ? i : -i; }

int f() {cout <<"call to function f"<<endl;}

void userCode(int &x) {
     int ans;
     cout << "UNSAFE CALLS"<<endl;
     cout << x << endl; ans = unsafe( x++ ); // X incremented twice!
     cout << x << endl; ans = unsafe( f() ); // f() called twice !

     cout << "SAFE CALLS"<<endl;
     cout << x << endl; ans = safe( x++ ); // X incremented only once : all OK
     cout << x << endl; ans = safe( f() ); // only one call
}
```

Recursivity

- Recursivity is the property that functions have to be called by themselves.
- For a classical (and not really efficient) example, to obtain the factorial of a number (n!) the mathematical formula would be:

n! = n * (n-1) * (n-2) * (n-3) ... * 1

- and a recursive function to calculate this in C++ could be:

Main program

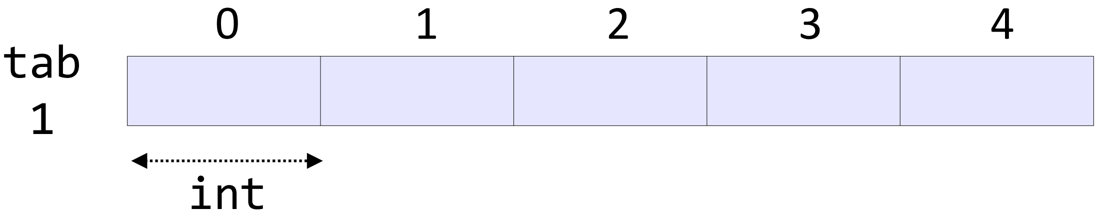
```
// factorial calculator
#include <iostream>
using namespace std;
//watchout the numerical limit…
long factorial (long a){
  if (a > 1)   return (a * factorial (a-1)); //recursion here
  else         return (1);
}

int main (){
  long number;
  cout << "Please type a number: ";
  cin >> number;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

Result

```
Please type a number:
9
9!  = 362880
```

Arrays

- Series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.
- For example, an array to contain 5 integer values of type int called `tab1` could be represented like this:



- where each blank panel represents an element of the array, in this an integer. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length.
- Like a regular variable, an array must be declared before it is used:

```
<type> <name> [<number of elements>];
// for this example: int tab1[5];
```

  - `<type>` is a valid type (like int, float...)
  - `<name>` is a valid identifier
  - `<number of elements>` always enclosed in square brackets [ ].

Array initialization

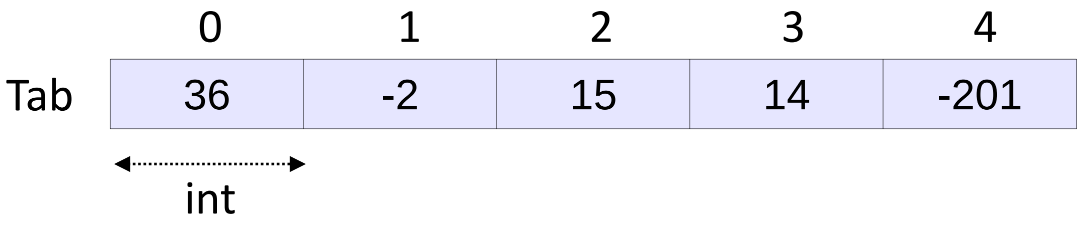
- When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, **its elements will not be initialized to any value by default**
    - Their content will be undetermined until we store some value in them.
    - The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.
    - Consequence: never forget to initialize all the values of the array

- In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }.

- Example:

```
int Tab [] = { 36, -2, 15, 14, -201 };
```

leads to the following array:



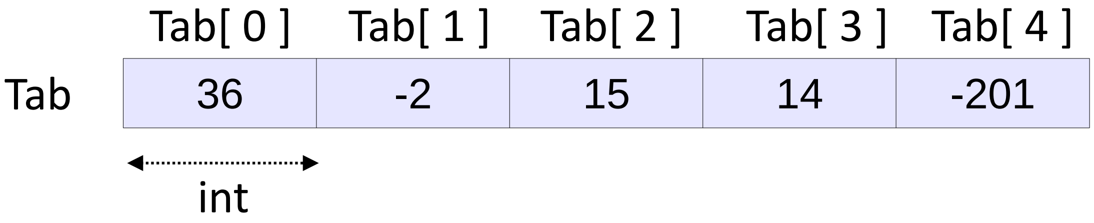
	0	1	2	3	4
Tab	36	-2	15	14	-201

Accessing the values of an array

- In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable (read and write access) using:

```
name[index]
```

- On our example, this means:



	Tab[0]	Tab[1]	Tab[2]	Tab[3]	Tab[4]
Tab	36	-2	15	14	-201

- To store the value 48 in the third element of Tab, we could write

```
Tab[2] = 48;
```

- To store the value of the third element of Tab into a variable called a, we could write

```
a = Tab[2];
```

- Note for Lab on matrices...See how "tricky" it is to implement the operator [ ] for home-made matrices

Accessing the values of an array

- Notice that
  - First element is `Tab[0]`, second element is `Tab[1]`, third one is `Tab[2]`, fourth one is `Tab[3]` and the last element is `Tab[4]`

- If you try `Tab[5]`, you would be accessing the sixth element of Tab and therefore exceeding the size of the array, which leads the program to crash
  - In C++ it is syntactically correct to exceed the valid range of indices for an array
  - Can (do) create problems, since accessing out-of-range elements do not cause compilation errors but causes runtime errors
  - The reason why this is allowed will be seen further ahead when we begin to use pointers

- Remember the two uses that brackets [ ] have for arrays
  - One is to specify the size of arrays when they are declared: `int Tab[5];`
  - The second one is to specify indices for concrete array elements:

```
Tab[1] = 28; // updating a value in the array
a = Tab[2];  // assigning a value to a variable a
```

Multidimensional arrays

- Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a bidimensional table made of elements, all of them of a same uniform data type.

- Example:

```
int Tab2[3][5]; //2-dimensional array declaration
```

- Leads to the following 2-dimensional array

Tab2[0][0]	Tab2[0][1]	Tab2[0][2]	Tab2[0][3]	Tab2[0][4]
Tab2[1][0]	Tab2[1][1]	Tab2[1][2]	Tab2[1][3]	Tab2[1][4]
Tab2[2][0]	Tab2[2][1]	Tab2[2][2]	Tab2[2][3]	Tab2[2][4]

Multidimensional arrays are not limited to two dimensions

- But be careful! The amount of memory needed for an array rapidly increases with each dimension. For example:

```
float tab3 [50][15][12][50]; // how many elements?
```

Multidimensional arrays

- Multidimensional arrays are just an abstraction for programmers, since we can obtain the same results with a simple array just by putting a factor between its indices:

```
int Tab4 [4][8];    // 4x8 integer array
int Tab5 [32];      // same size, but 1-dimensional array
```

- With multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bidimensional array and the other one uses a simple array:

multidimensional array	pseudo-multidimensional array

#define WIDTH 5
#define HEIGHT 3
 	
int tab5 [HEIGHT][WIDTH];
int n, m;
 	
int main () {
  for (n = 0; n < HEIGHT; n++)
    for (m = 0; m < WIDTH; m++)
    {
      tab5[n][m] = (n+1)*(m+1);
    }
  return 0;
}
	#define WIDTH 5
	#define HEIGHT 3
	
	int tab6 [HEIGHT * WIDTH];
	int n, m;
	
	int main () {
	  For (n = 0; n < HEIGHT; n++)
	    For (m = 0; m < WIDTH; m++)
	    {
	     tab6[n*WIDTH+m] = (n+1)*(m+1);
	    }
	 return 0;
	}

Arrays as parameters

- At some moment we may need to pass an array to a function as a parameter.
- In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice: almost the same effect and much more efficient operation.

- To accept arrays as parameters
  - Specify in the parameters:
    - the element type of the array
    - an identifier and a pair of void brackets [ ]
  - For example, the following function : `void procedure (int arg[]);` accepts a parameter of type "array of int" called `arg`. In order to pass to this function an array declared as `int myarray [40];` it would be enough to write a call like this: `procedure (myarray);`

Example

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length)
{
  for (int n=0; n<length; n++)
    cout << arg[n] << " ";
  cout << "\n";
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
  return 0;
}
```

The first parameter `(int arg[])` accepts any array whose elements are of type int, whatever its length.

So we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter.

This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

Multidimensional arrays as parameters

In a function declaration, it is possible to include multidimensional arrays as parameters.

The idea here is simply to consider multidimensional arrays as arrays of arrays.

For instance, the format for a tridimensional array parameter passed as a parameter would be:

```
base_type[][depth][depth]
```

Notice that the first brackets [ ] are left blank while the following ones are not: this can be seen as a an array of unknown size whom elements are bidimensional arrays of size depth x depth. For example, a function with a multidimensional array as argument could be:

```
void procedure ( int myarray[][3][4] ); // [] for the first dimension
```

The compiler must be able to determine within the function what is the size of the elements of the array. Arrays, simple or multidimensional, passed as function parameters are a quite common source of errors for novice programmers.

See lecture about pointers for a better understanding (or not) on how arrays operate.

Pointers

- Up to now: variables
  - memory cells
  - accessed using their identifiers
  - did not care about the physical location within memory

- The memory: succession of memory cells, each one of the minimal size that computers manage (one byte).
  - These single-byte memory cells are numbered in a consecutive way
  - Cells can be located in the memory
    - **unique** address
    - memory cells follow a successive pattern

- For example, if we are looking for cell 1776 we know that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and so on.

Reference operator &

- When we declare a variable
  - The amount of memory needed is assigned at a specific location in memory (its memory address)
  - We do not actively decide the exact location

- It may be interesting to know the address where our variable is being stored during runtime in order to operate with relative positions to it (instead of handling a huge variable)

- The address that locates a variable within memory is called a reference to that variable
  - Can be obtained by preceding the identifier of a variable with an ampersand sign & (reference operator)
  - can be literally translated as "address of "

- For example:

```
MyVarAddress = &MyVar;
// MyVarAddress is the address of the variable MyVar
```

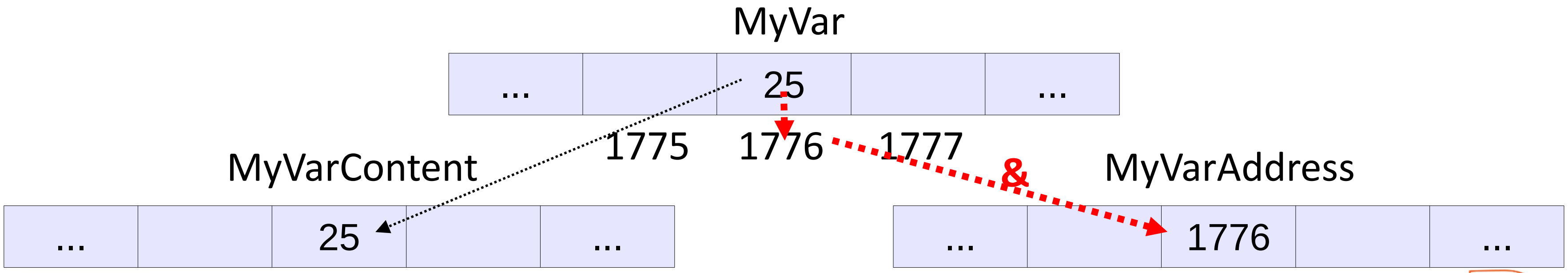
- Assign to `MyVarAddress` the address of the variable `MyVar`
  - No longer talking about the content of the variable itself, but about its reference (i.e., its address in memory)

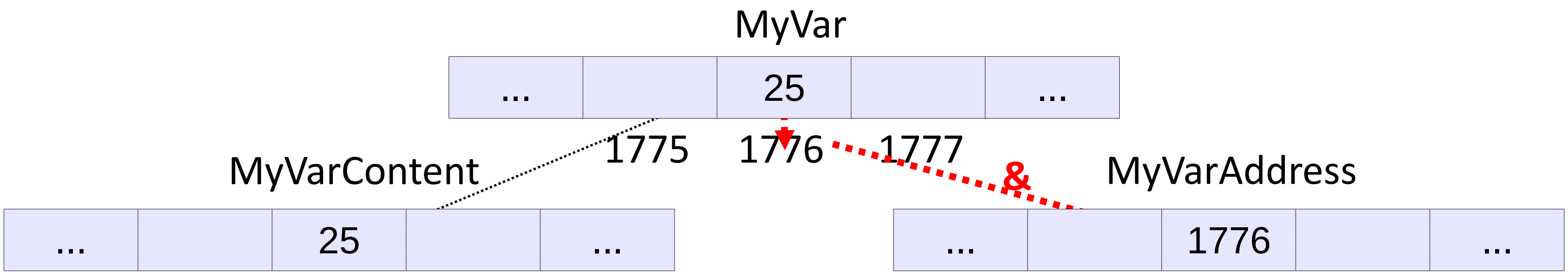
- From now on we are going to assume that MyVar is placed during runtime in the memory address 1776.
  - 1776 is just an arbitrary assumption
  - we cannot know before runtime the real value the address of a variable will have in memory

- Consider the following code fragment:

```
MyVar = 25;
MyVarContent = MyVar;
MyVarAddress = &MyVar;
```

- In the memory, this corresponds to:





MyVar

...		25		...
	1775	1776	1777	

MyVarContent

...		25		...

&

MyVarAddress

...		1776		...

Step by step

- we have assigned the value 25 to `MyVar`, stored at the address 1776

```
MyVar = 25;
```

- Perform a copy of the content of `MyVar` to `MyVarContent`

```
MyVarContent = MyVar;
```

- Finally, the third statement copies to `MyVarAddress` not the value contained in `MyVar` but a reference to it by

```
MyVarAddress = &MyVar;
```

- A variable that stores the reference to another variable is called a **pointer**

- Pointers are often a source of confusion, so take the time to
  - understand
  - practice!

Dereference operator *

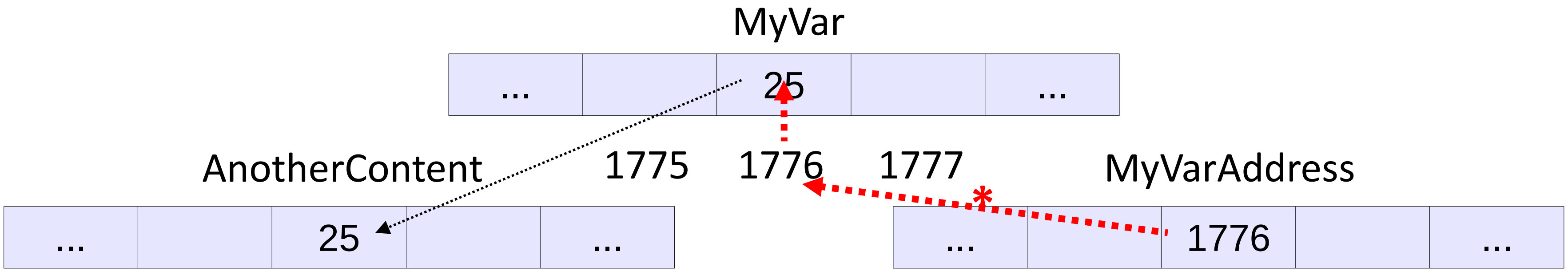
- Pointers "point to" the variable whose reference they store
- Using a pointer we can directly access the value stored in the variable which it points to
  - simply precede the pointer's identifier with an asterisk *
    - acts as dereference operator
    - can be literally translated to "value pointed by"
- Therefore, if we write:

```
AnotherContent = *MyVarAddress;
```

- This statement could be read as:

"AnotherContent is assigned to the value pointed by MyVarAddress"

- AnotherContent would take the value 25, since MyVarAddress is 1776, and the value pointed by 1776 is 25.



Declaring variables of pointer types

- Types matter!
  - Not the same thing to point to a char as to point to an `int` or a `float`
- Syntax

```
type * name;
```

  - `type` is the data type of the value that the pointer is intended to point t o
  - `type` is not the type of the pointer itself! For example:

```
int * number; char * character; float * greatnumber;
```

  - Each one is intended to point to a different data type
  - All of them are pointers and will occupy the same amount of space in memory (depends on the platform)
  - The data to which they point to do not occupy the same amount of space nor are of the same type

- Although these three example variables are all of them pointers which occupy the same size in memory
  - they are said to have different types: int*, char* and float* , respectively

Example

Remember: the asterisk sign (*) used when declaring a pointer only means that it is a pointer and should not be confused with the dereference operator.

Main program

```
// my first pointer
#include <iostream>
using namespace std;

int main (){
  int firstvalue, secondvalue;
  int * mypointer;  // * as pointer declaration

  mypointer = &firstvalue;
  *mypointer = 10;  // * as dereference operator
  mypointer = &secondvalue;
  *mypointer = 20;  // same here
  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

Result

```
firstvalue is 10
secondvalue is 20
```

Example

Main program

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

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue

  *p1 = 10;          // value pointed by p1 = 10

  *p2 = *p1;         // value pointed by p2 = value pointed by p1

  p1 = p2;           // p1 = p2 (value of pointer is copied)

  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

Result

```
firstvalue is
10
secondvalue is
20
```

Pointers and Arrays

- The identifier of an array is equivalent to the address of its first element
  - A pointer is equivalent to the address of the first element that it points to
  - They are the same concept in C++
  - For example, supposing these two declarations:

    ```
    int myarray [20];
    int * p;
    ```

  - The following assignment operation would be valid:

    ```
    p = myarray;
    ```

- The only difference is that we could change the value of pointer p by another one, whereas `numbers` will always point to the first of the 20 elements of type int with which it was defined.
  - Therefore, unlike p, which is an ordinary pointer, `numbers` is an array, and an array can be considered a constant pointer.
  - This implies that the following allocation would not be valid:

    ```
    numbers = p;
    ```

Another example

Main program

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];  // array, const pointer
  int * p;         // pointer to int

  p = numbers;     // pointer assigned to the first
                   // element of numbers

  *p = 10;         // value pointed by p is set to 10

  p++;             // go one block further in the memory
  *p = 20;         // value pointed by p is set to 10

  p = &numbers[2];  *p = 30; //p is set to address of 3rd elt

  p = numbers + 3;  *p = 40; //p is set to address of 1st + 3blocks

  p = numbers;  *(p+4) = 50; //p is set to address of 1st and
                             //the content of the 5th block is set to 50

  //display values contained in array numbers
  for (int n=0; n<5; n++)    cout << numbers[n] << ", ";
  return 0;
}
```

Result

```
10, 20, 30, 40, 50,
```

Discussion

- In the chapter about arrays we used brackets [ ] several times in order to specify the index of an element of the array to which we wanted to refer
  - These bracket sign operators [ ] are also a dereference operator known as offset operator
  - They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced using pointers arithmetic. For example:

```
a[5] = 0;          // a [offset of 5] = 0
*(a+5) = 0;        // pointed by (a+5) = 0
```

- These two expressions are equivalent and valid if **a** is a pointer or if **a** is an array

Pointer Initialization

- When declaring pointers we may want to explicitly specify which variable we want them to point to:

```
int number;
int *pnumber = &number;
```

- When a pointer initialization takes place
  - Assign the reference value to where the pointer points (pnumber), never the value being pointed (*pnumber)

- At the moment of declaring a pointer, the asterisk (*) indicates only that it is a pointer, it is not the dereference operator. You cannot dereference a pointer if it has not been initialized correctly.

- Thus, we must take care not to confuse the previous code with:

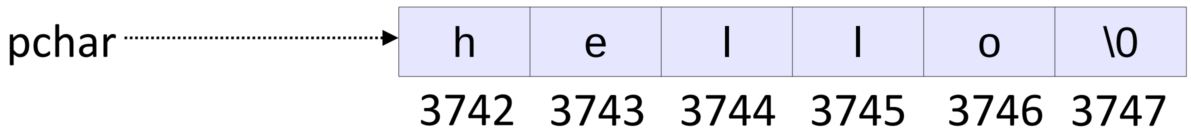
```
int number;
int *pnumber;
*pnumber = &number; // wrong! Compilation error
*pnumber = 12; //segmentation fault. pnumber not initialized
```

Char pointers

- As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

```
char * pchar = "hello"; //notice the  “ “ syntax
```

- Some memory space is reserved to contain "hello"
- A pointer to the first character of this memory block is assigned to pchar
- If we imagine that "hello" is stored at the memory locations that start at addresses 3742, we can represent the previous declaration as:



pchar contains the value 3742 and not 'h' nor "hello"

- The pointer pchar points to a sequence of characters and can be read as if it was an array
  - For example:

    ```
    *(pchar+4) or pchar[4] // access to the 5th elt
    ```
  - Both expressions have a value of 'o'

Pointers arithmetic

- Arithmetical operations on pointers
  - Only addition and subtraction
  - Different behavior with pointers according to the size of the data type to which they point

- As previously seen,the different fundamental data types occupy more or less space than others in the memory.
  - Let's assume that in a given compiler for a specific machine, char takes 1 byte, short takes 2 bytes and long takes 4 bytes
  - Suppose that we define three pointers in this compiler:

```
char *mychar;
short *myshort;
long *mylong;
```

  - And that we know that they point to memory locations 1000, 2000, and 3000, respectively

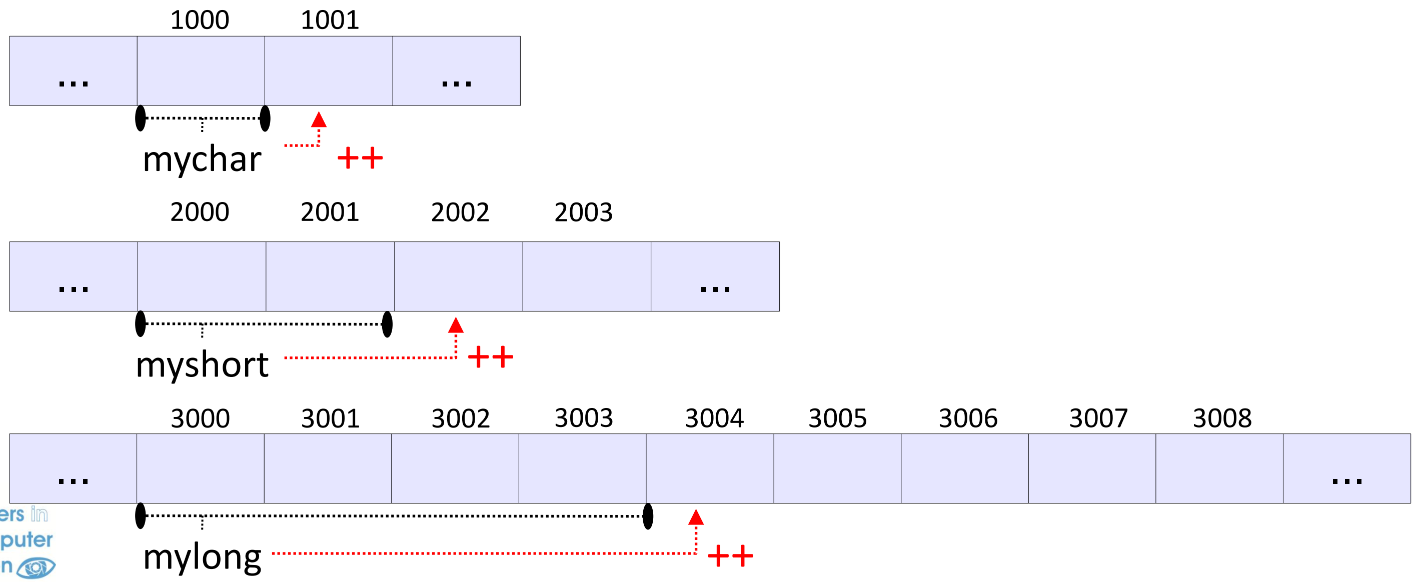
Pointers arithmetic

- So if we write:

```
mychar++; myshort++; mylong++;
```

  - `mychar` would contain the value 1001
  - `myshort` would contain the value 2002
  - `mylong` would contain 3004

- The reason is that when adding 1 to a pointer we are making it to point to the following element of the same type
  - the size (in bytes) of the type pointed is added to the pointer.



Pointers arithmetic

- This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

- Both the increase (++) and decrease (--) operators have greater operator precedence than the dereference operator (*)

- Both operators have a special behavior when used as suffix
  - the expression is evaluated with the value it had before being increased.
  - Therefore, the following expression may lead to confusion:

```
*p++ //which operator is applied first?? * or ++
```

- Because ++ has greater precedence than *, this expression is equivalent to *(p++).

Pointers arithmetic

- Notice the difference with: (*p)++
  - The expression would have been evaluated as the value pointed by p increased by one.
  - The value of p (the pointer itself) would not be modified
  - What is being modified is what it is being pointed to by this pointer
- If we write: *p++ = *q++;
  - Because ++ has a higher precedence than *, both p and q are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to *p is *q before both p and q are increased. And then both are increased. It would be roughly equivalent to:

```
*p = *q;
++p;
++q;
```

- Suggestion: use parentheses ( ) when you are unsure

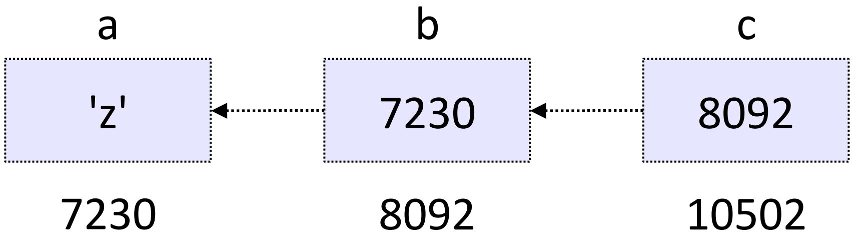
Pointers to pointers

- C++ allows the use of pointers that point to pointers
  - Useful for arrays of arrays, i.e. multidimensional arrays

- Add an asterisk * for each level of reference in their declarations:

```
char a; char * b; char ** c;
a = 'z'; b = &a; c = &b;
```

- This, supposing the randomly chosen memory locations for each variable of 7230, 8092, and 10502 could be represented as:



  - c has type char** and a value of 8092
  - *c has type char* and a value of 7230
  - **c has type char and a value of 'z'

Void pointers

- The void type of pointer is a special type of pointer
  - represents the absence of type
  - void pointers are pointers that point to a value that has no type
    - also an undetermined length and dereference properties

- This allows void pointers to point to any data type, from an integer value or a float to a string of characters

- Limitation:
  - the data pointed by them cannot be directly dereferenced
  - which is logical, since we have no type to dereference to

- Therefore, we will always have **to cast** the address in the void pointer to some other pointer type that points to a concrete data type before dereferencing it, see example next

- One of its uses may be to pass generic parameters to a function
- Other usage, pointers to functions

Main program

```
// increaser function with void* pointer
#include <iostream>
using namespace std;

void increase (void* data, int psize) {

  if ( psize == sizeof(char) )   {
    char* pchar;
    pchar = (char*)data;
    ++(*pchar);
  }

  else if (psize == sizeof(int) )  {
    int* pint;
    pint = (int*)data;
    ++(*pint);
  }
}

int main () {
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << endl;
  return 0;
}
```

Result

```
y, 1603
```

Null pointers

- A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address

- This value is the result of type-casting the integer value zero to any pointer type.

```
int * p;
p = 0;     // p has a null pointer value
p = NULL;  // is equivalent
```

- Do not confuse null pointers with void pointers
  - A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere"
  - A void pointer is a special type of pointer that can point to somewhere without a specific type.
  - One refers to the value stored in the pointer itself and the other to the type of data it points to.

Pointers to functions

- The typical use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced
- In order to declare a pointer to a function we have to declare **it like the prototype of the function** except that **the name of the function is enclosed between parentheses** ( ) and **an asterisk * is inserted before the name**:

Main program : `pointer to functions`

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

int addition (int a, int b)    { return (a+b); }
int subtraction (int a, int b) { return (a-b); }

int operation (int x, int y, int (*functocall)(int,int)){
  int g = (*functocall)(x,y);
  return (g);
}

int main (){
  int m,n;
//ptr to any fction with a signature as : int f (int, int)
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

Result

```
8
```

Dynamic Memory

- Until now, in all our programs, we have only had as much memory available as we declared for our variables

- In other words, **we knew a priori the size of our arrays**, etc, before the execution of the program

- But, what if we need an **unknown amount of memory** that **can only be determined during runtime**?
  - For example, in the case where we need some user input to determine the necessary amount of memory space

- The answer is dynamic memory, for which C++ integrates the operators **`new`** and **`delete`**

Operators new and new[ ]

- In order to request dynamic memory we use the operator new
- new is followed by a data type specifier and -if a sequence of more than one element is required- the number of these within brackets []
- It returns a pointer to the beginning of the new block of memory allocated

- Syntax

```
pointer = new type;
pointer = new type [number_of_elements];
```

  - The first expression is used to allocate memory to contain one single element of type type.
  - The second one is used to assign a block (an array) of elements of type type, where number_of_elements is an integer value

```
int * p1 = new int;      //to allocate one integer
int * p2 = new int [5]; //to allocate an array of 5 integers
```

  - In this case, the system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence

```
int * p1 = new int;           //to allocate one integer
int * p2 = new int [5];       //to allocate an array of 5 integers
```

- The first element pointed by `p2` can be accessed either with the expression `p2[0]` or the expression `*p2`.
- The second element can be accessed either with `p2[1]` or `*(p2+1)` and so on
- Difference between declaring a normal (static) array and assigning dynamic memory to a pointer
  - The size of an array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution
  - The dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size
- The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted.
  - It is important to have some mechanism to check if our request to allocate memory was successful or not

Operator new and new[ ]

- C++ provides two standard methods to check if the allocation was successful

- Handling exceptions
  - Using this method an exception of type bad_alloc is thrown when the allocation fails
  - Exceptions are a powerful C++ feature explained later.
  - For now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated
  - This exception method is the default method used by new, and is the one used in a declaration like:

    ```
    p2 = new int [5];  // if it fails an exception is thrown
    ```

- Nothrow
  - When a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution

- This method can be specified by using a special object called nothrow, declared in header <new>, ex:

```
int *myarray = new (nothrow) int [5]; // returns 0 if alloc fails
```

Operator new and new[ ]

- In this case, if the allocation of this block of memory failed, the failure could be detected by checking if `MyPointer` took a null pointer value:

```
int * MyPointer = new (nothrow) int [5];

if (MyPointer == 0) {// error assigning memory.
  cout<<"memory allocation failed"<<endl;
  exit(1);
}
```

- `nothrow` requires more work than the exception method, since the value returned has to be checked after each and every memory allocation

- Anyway this method can become tedious for larger projects, where the exception method is generally preferred (further lectures)

Operators delete and delete[]

- Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it MUST be freed so that the memory becomes available again for other requests of dynamic memory

- Operator `delete`

```
delete pointer;     //single element deletion
delete [] pointer;  //deletion of an array
```

- The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements

- The value passed as argument to `delete` must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

Main program

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

int main (){

  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;

  p= new (nothrow) int[i]; //allocate array

  if (p == 0)    cout << "Allocation error"<<endl;

  else  {   //ask values to enter into the array
    for (n=0; n<i; n++)    {
      cout << "Enter number: ";
      cin >> p[n];
    }

    //display the array
    cout << "You have entered: ";
    for (n=0; n<i; n++)     cout << p[n] << ", ";

   //our job is done, now free the memory
    delete[] p;
  }
  return 0;
}
```

Result

```
How many numbers would you
like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436,
1067, 8, 32,
```

Operators delete and delete[]

- Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant value:

```
p = new (nothrow) int[i]; // i is a variable
```

- But the user could have entered a value for i so big that our system could not handle it : for instance, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case

- Remember that in the case that we tried to allocate the memory without specifying the `nothrow` parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

- It is a good practice to always check if a dynamic memory block was successfully allocated
  - If you use the `nothrow` method, you should always check the value of the pointer returned
  - Otherwise, use the exception method, even if you do not handle the exception
  - This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not

Structures

- Array = agglomeration of data of same type
- Structure = agglomeration of data of potentially different types

- A data structure is a group of data elements grouped together under one name. These data elements, known as **members**, can have different types and different lengths.

- Data structures are declared in C++ using the following syntax:

```
struct structure_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    .
    .
} object_names;
```

- `structure_name` is a name for the structure type
- `object_name` can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a type and a valid identifier as its name

Structures

- A data structure creates a new type
  - Once a data structure is declared, a new type is created and can be used in the rest of the program as.
  - For example:

```
struct product {
  int weight;
  float price;
} ;

//declare variables of type product:
 product apple;
 product banana, melon;
```

  - We have first declared a structure type called `product` with two members: weight and price, each of a different fundamental type.
  -
  - We have then used this name of the structure type (product) to declare three objects of that type: `apple,` `banana` and `melon` as we would have done with any fundamental data type.

Structures

- Right at the end of the struct declaration, and before the ending semicolon, we can use the optional field `object_name` to directly declare objects of the structure type.

- For example, we can also declare the structure objects apple, banana and melon at the moment we define the data structure type this way:

```
struct product {
  int weight;
  float price;
} apple, banana, melon;
```

- It is important to clearly differentiate
  - what is the structure type name
  - what is an object (variable) that has this structure type.
  - we can instantiate many objects (i.e. variables, like apple, banana and melon) from a single structure type (product).

Structures

- Once we have declared our three objects of a determined structure type (apple, banana and melon) we can operate directly with their members

- Use a dot (.) inserted between the object name and the member name

- For example, we could operate with any of these elements as if they were standard variables of their respective types:

```
apple.weight // access to the member weight of the variable apple
apple.price  // access to the price weight of the variable apple
banana.weight
banana.price
melon.weight
melon.price
```

- Each one of these has the data type corresponding to the member they refer to: apple.weight, banana.weight and melon.weight are of type int, while apple.price, banana.price and melon.price are of type float.

Structures

Main program

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

struct movies_t {
  string title;
  int year;
} mine, yours;

void printmovie (movies_t movie); // function for display

int main (){
  mine.title = "Shrek";
  mine.year = 2001;

  cout << "Enter title: "; cin >> yours.title;
  cout << "Enter year: ";  cin >> yours.year;

  cout << "My favorite movie is:"<<endl;  printmovie (mine);
  cout << "And yours is:"<<endl;          printmovie (yours);
  return 0;
}

void printmovie (movies_t movie){
  cout << movie.title << " (" << movie.year << ")"<<endl;
}
```

Result

```
Enter title: Alien
Enter year: 1979

My favorite movie is:
Shrek (2001)
And yours is:
Alien (1979)
```

Discussion

- As is, the following example is not going to work for film titles with several words

- The example shows how we can use the members of an object as regular variables. For example, the member `yours.year` is a valid variable of type int, and `mine.title` is a valid variable of type string

- The objects mine and yours can also be treated as valid variables of type `movies_t`, for example we have passed them to the function `printmovie(movies_t m)` as we would have done with regular variables

- Advantages of data structures: we can either refer to their members individually or to the entire structure as a block with only one identifier

- Data structures are a feature that can be used to represent databases, especially if we consider the possibility of building arrays of them

Main program : array of structures

```
#include <iostream>
#include <string>  // to handle strings
#include <sstream> // to handle string stream
using namespace std;
#define N_MOVIES 3

struct movies_t {
  string title;
  int year;
} films [N_MOVIES]; // now we declare an array of structures

void printmovie (movies_t movie); // display function

int main ( ) {
  string mystr; int n;
  for (n=0; n<N_MOVIES; n++)  {
    cout << "Enter title: ";
    getline (cin,films[n].title); // note the difference here
    cout << "Enter year: ";
    getline (cin,mystr);          // same use of getline
    stringstream(mystr) >> films[n].year; }

  cout << endl<<"You have entered these movies:"<<endl;
  for (n=0; n<N_MOVIES; n++)    printmovie (films[n]);
  return 0;
}

void printmovie ( movies_t movie ) {
  cout << movie.title;
  cout << " (" << movie.year << ") "<<endl;
}
```

Result

```
Enter title: Blade Runner
Enter year: 1982
Enter title: Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

You have entered these
movies:
Blade Runner (1982)
Matrix (1999)
Taxi Driver (1976)
```

Pointers to structures

- Like any other type, structures can be pointed by its own type of pointers:

```
struct movies_t {
  string title;
  int year;
};

movies_t amovie;   // variable amovie of type movies_t
movies_t * pmovie;// pointer to objects of type movies_t
```

- Here `amovie` is an object of structure type `movies_t`, and `pmovie` is a pointer to point to objects of structure type `movies_t`

- So, the following code would be valid

```
pmovie = &amovie;
```

- The value of the pointer pmovie would be assigned to a reference to the object amovie (its memory address)

- **To access the members of a pointer, use the operator -> instead of . (dot)**

Main program

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie; // variable of type movies_t
  movies_t * pmovie; //pointer to a movies_t
  pmovie = &amovie;  //pointer =  address of variable, as usual

  cout << "Enter title: ";
  getline (cin, pmovie->title); // notice the operator ->
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year; // same here

  cout << endl<< "You have entered:"<<endl;
  cout << pmovie->title;
  cout << " (" << pmovie->year << ") "<<endl;

  return 0;
}
```

Result

```
Enter title: Mars attacks
Enter year: 1996

You have entered:
Mars attacks (1996)
```

The operator ->

- Dereference operator

- Used exclusively with pointers to objects with members
  - Serves to access a member of an object to which we have a reference.
  - In the example we used:

    `pmovie->title`

- Which is for all purposes equivalent to `(*pmovie).title`

- Both expressions `pmovie->title` and `(*pmovie).title` are valid and both mean that we are evaluating the member title of the data structure pointed by a pointer called `pmovie`.
  - It must be clearly differentiated from:

    *pmovie.title
  - which is equivalent to:

    `*(pmovie.title)`

Reminder

This very often leads to confusions and errors, so keep it simple:

- void typedefs with pointers, unless you really got used to them
- use parentheses when unsure:

expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed by a	(*a).b
*a.b	Value pointed by member b of object a	*(a.b)

Nested structures

- Structures can also be nested. For instance:

```
struct movies_t {
  string title;
  int year; };

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
  } friend1, friend2;

friends_t * pfriends = &friend1;
```

- After the previous declaration we could use any of the following expressions:

```
friend1.name
friend2.favorite_movie.title
friend1.favorite_movie.year
pfriends->favorite_movie.year  // this is equivalent
```

Defined data types (typedef)

- Allows the definition of our own types based on other existing data types. We can do this using the keyword typedef, whose format is:

```
typedef <existing_type> <new_type_name> ;
```

- Where `existing_type` is a C++ fundamental or compound type and `new_type_name` is the name for the new type we are defining. For example:

```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

- In this case we have defined four data types: C, WORD, pChar and field as char, unsigned int, char* and char[50], respectively, that we could perfectly use in declarations later as any other valid type:

```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

typedef

- Does not create different types
- Only creates synonyms of existing types.
  - That means that the type of myword can be considered to be either WORD or unsigned int
- Can be useful to define an alias for a type that is frequently used within a program
- Can be useful to define types when it is possible that we will need to change the type in later versions of our program, or if a type you want to use has a name that is too long or confusing

Enumeration

- Enumerations create new data types to contain something different that is not limited to the values fundamental data types may take.
- Syntax

```
enum enumeration_name {
  value1,
  value2,
  value3,
...
} object_names;
```

For example:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

- Notice that we do not include any fundamental data type in the declaration

- We have created a whole new data type from scratch without basing it on any other existing type

Enumeration

- The possible values that variables of this new type `color_t` may take are the new constant values included within braces

- For example, once the enumeration is declared the following expressions will be valid:

```
colors_t mycolor;
mycolor = blue;
if (mycolor == green) mycolor = red;
```

- Enumerations are type compatible with numeric variables, so their constants are always assigned an integer numerical value internally

- If it is not specified, the integer value equivalent to the first possible value is equivalent to 0 and the following ones follow a +1 progression

- We can explicitly specify an integer value for any of the constant values that our enumerated type can take. For example:

```
enum months_t { january=1, february, march, april,  may, june, july, august,
    september, october, november, december} year;
```

- In this case, variable year of enumerated type `months_t` can contain any of the 12 possible values that go from january to december and that are equivalent to values between 1 and 12 (not between 0 and 11, since we have made january equal to 1).