Fundamentals of Programming I

3 Types and Instructions II

Grado en Ingeniería Informática

Luis Hernández Yáñez
Facultad de Informática
Universidad Complutense

Index

Types, Values and Variables

Luis Hernández Yáñez

Types, Values and Variables

Type A value set with its possible operations

Value Group of bits interpreted as data of a specific type

Variable (or constant)
Certain memory with a name for holding values of some type

Declaration Instruction that identifies a name

Definition Declaration that assigns memory to a variable or constant

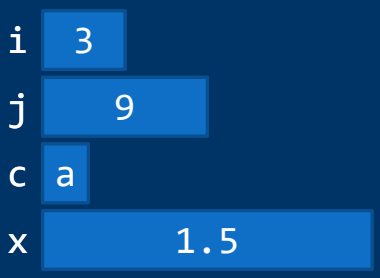
Luis Hernández Yáñez

BY NC SA

Variables

Enough memory for each type of values

```
short int i = 3;
int j = 9;
char c = 'a';
double x = 1.5;
```



The meaning of the bits depends on the variable type:

`00000000 00000000 00000000 01111000`

- Interpreted as `int` it is the integer `120`
- Interpreted as `char` (only `01111000`) it is the character `'x'`

Types

- ✓ Simple
  - Standard: `int`, `float`, `double`, `char`, `bool`
    Predefined set of values
  - Declared by the user: *enumerated*
    Set of values defined by the programmer
- ✓ Structured
  - Homogeneous collections: *arrays* (`string`)
    All the elements of the same type
  - Heterogeneous collections: *structures*
    Elements with different types

Simple Standard Types

With their possible modifiers:

```
[unsigned] [short] int
long long int
float
[long] double
char
bool
```

`long int` ≡ `int`

Variable definition:

```
type name [ = expression] [, ...];
```

Named constant definition:

```
const type name = expression;
```

Fundamentals of Programming I

Type Conversion

Automatic Type Conversion

Type Promotion

Operands of different (compatible) types:

The value of the *lesser* type is promoted to the *greater* type

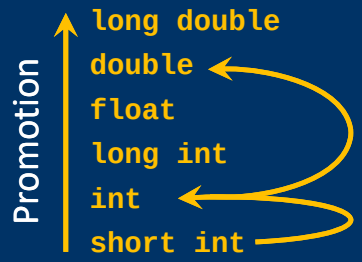
```
short int i = 3;
int j = 2;
double a = 1.5, b;
b = a + i * j;
```

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

→ `short int` value `3` (2 bytes) → `int` (4 bytes)

```
b = 1.5 + 6;
```

→ `int` value `6` (4 bytes) → `double` (8 bytes)



Secure and Non-Secure Conversions

Secure conversion:

From a lesser type to a greater type

`short int` → `int` → `long long int` → ...

Non-secure conversion:

From a greater type to a lesser type

```
int integer = 1234;
char character;
character = integer; // Non secure conversion
```

Less memory: Loss of information in the conversion process

```
long double
double
float
long long int
int
short int
```

Casts

Force a type conversion:

type(*expression*)

The *expression* result is treated as that *type* value

```
int a = 3, b = 2;
cout << a / b;          // Prints 1 (integer division)
cout << double(a) / b; // Prints 1.5 (real division)
```

Highest priority

Luis Hernández Yáñez

Fundamentals of Programming I

User-Declared Types

Luis Hernández Yáñez

User-Declared Types

We describe the values for the type variables

```
typedef description type_name;
```

Valid identifier

Names for our own types:
Lowercase `t` followed by one or more capitalized words
We will color them in orange to emphasize they are types

```
typedef description tMyType;
typedef description tCoin;
typedef description tTypesOfQualificacion;
```

Type declaration vs. variable definition

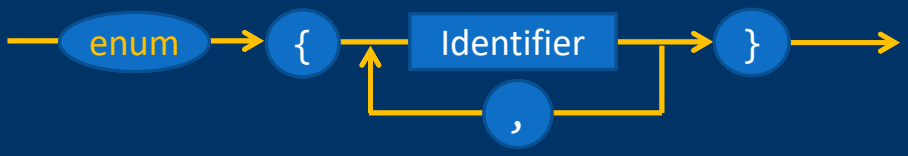
Fundamentals of Programming I

Enumerated Types

Enumerations

Enumeration of the set of valid values for the variables:

```
enum { symbol1, symbol2, ..., symbolN }
```




```
enum { cent, two_cents, five_cents, ten_cents,
       twenty_cents, half_euro, euro }
```

Symbols are the literal values (yellow) the variables can get

Luis Hernández Yáñez

Enumerated Types

Enumerated types improve legibility

```
typedef description type_name;
```

We choose a name for the type: `tCoin`

description

```
typedef enum {cent, two_cents, five_cents, ten_cents,
              twenty_cents, half_euro, euro } tCoin;
```

In the declaration scope a new type is recognized: `tCoin`

```
tCoin coin1, coin2;
```

Each variable of this type will hold any of those symbols

```
coin1 = two_cents;
coin2 = euro;
```

coin1	two_cents
coin2	euro

(Internally, integers are used)

Luis Hernández Yáñez

Input/Output for Enumerated Types

```
typedef enum { January, February, March, April, May, June,
July, August, September, October, November, December } tMonth;
tMonth month;
```

Reading a value for `month` variable: `cin >> month;`

An integer value is expected...

We can't directly write `January` or `June`

If we write the variable's value on the screen:

`cout << month;`

An integer value will be shown...

→ Specific input/output code

Reading Values for Enumerated Types

```
typedef enum { January, February, March, April, May, June,
   July, August, September, October, November, December } tMonth;
```

```
int op;
cout << " 1 - January"   << endl;
cout << " 2 - February"  << endl;
cout << " 3 - March"     << endl;
cout << " 4 - April"     << endl;
cout << " 5 - May"       << endl;
cout << " 6 - June"      << endl;
cout << " 7 - July"      << endl;
cout << " 8 - August"    << endl;
cout << " 9 - September" << endl;
cout << "10 - October"   << endl;
cout << "11 - November"  << endl;
cout << "12 - December"  << endl;
cout << "Month number: ";
cin >> op;
tMonth month = tMonth(op - 1);
```

Cast
`int` → `tMonth`

Writing Enumerated Types Values

```
typedef enum { January, February, March, April, May, June,
July, August, September, October, November, December } tMonth;
```

```
if (month == January)
   cout << "January";
if (month == February)
   cout << "February";
if (month == March)
   cout << "March";
if (month == April)
   cout << “April";
...
if (month == December)
   cout << "December";
```

We can also use a `switch` instruction

Enumerated Types

Ordered set of values (position in enumeration)

```
typedef enum { Monday, Tuesday, Wednesday, Thursday,
   Friday, Saturday, Sunday } tDayOfWeek;

tDayOfWeek day;
...
if (day == Thursday)...
bool notWeekDay = (day >= Sunday);
```

`Monday < Tuesday < Wednesday < Thursday < Friday < Saturday < Sunday`

There are no increment and decrement operators → Emulation with casts:

```
int i = int(day); // day shouldn't have Sunday value!
i++;
day = tDayOfWeek(i);
```

Enumerated Types: Example

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

typedef enum { January, February, March, April, May, June, July,
               August, September, October, November, December } tMonth;
typedef enum { Monday, Tuesday, Wednesday, Thursday, Friday,
               Saturday, Sunday } tDayOfWeek;

string strMonth(tMonth month);
string strDay(tDayOfWeek day);

int main() {
   tDayOfWeek today = Monday;
   tMonth month = November;
   int day = 4, year = 2013;
```

If types are used in several functions, declare them before the prototypes

.../...

Enumerated Types: Example

```
   // Print the date...
   cout << "Today is: " << strDay(today) << ", "
      << strMonth(month) << " " << day << ", " << year << endl;

   cout << "Past midnight..." << endl;
   day++;
   int i = int(today);
   i++;
   today = tDayOfWeek(i);

   // Print the date...
   cout << "Today is: " << strDay(today) << ", "
      << strMonth(month) << " " << day << ", " << year << endl;

   return 0;
}
```

.../...

Enumerated Types: Example

dates.cpp

```
string strMonth(tMonth month) {
   string str;

   if (month == January)
      str = "January";
   if (month == February)
      str = "February";
   ...
   if (month == December)
      str = "December";

   return str;
}
```

```
string strDay(tDayOfWeek day) {
   string str;

   if (day == Monday)
      str = "Monday";
   if (day == Tuesday)
      str = "Tuesday";
   ...
   if (day == Sunday)
      str = "Sunday";

   return str;
}
```


Fundamentals of Programming I

Input/Output with Text Files

Files

Program data: in main memory (volatile)

Media (devices) for permanent storage:

- Magnetic disks, fixed (internal) or portable (external)
- Magnetic tapes
- Optical disks (CD, DVD, BlueRay)
- USB memories

...

Information is kept in files

Data sequences

Text Files and Binary Files

Text File: character sequence

T	o	t	a	l	:		1	2	3	.	4	↵	A	…

Binary File: binary code sequence

A0	25	2F	04	D6	FF	00	27	6C	CA	49	07	5F	A4	…

(Codes represented in hexadecimal notation)

Files are managed in programs by means of *streams*

Text files: *text streams*

Text file I/O is similar to console I/O

Text Files

Texts arranged in several lines: End-of-line character (Enter) between lines

Possibly several data in the same line

Example: Company sales

In each line, client's NIF, units bought, unit price and product description, separated by one space

```
12345678F 2 123.95 DVD Player↲
00112233A 1 218.4 Portable Disk↲
32143567J 3 32 16Gb USB Memory↲
76329845H 1 134.5 ADSL Modem↲
...
```

Normally finished with special data (*sentinel*): For example with `X` as NIF

Streams for Text Files

`#include <fstream>`

✓ Reading from file: input stream

✓ Writing to file: output stream

We can't read and write at the same time

A text stream can be used to read or to write:

— Input streams (files) : variables of type `ifstream`

— Output streams (files) : variables of type `ofstream`

`fstream` library

Reading from Text Files

Luis Hernández Yáñez

Input Text Streams

`ifstream`

Reading from a text file:

1. Declare a variable of type `ifstream`
2. Associate the variable with the text file (*open the file*)
3. Perform read operations
4. Dissociate the variable from the text file (*close the file*)

Luis Hernández Yáñez

cc BY NC SA

Reading from Text Files

Opening the file

Connects the variable with the text file

```
stream.open(literal_string);
```

```
ifstream file;
file.open("abc.txt");
if (file.is_open()) ...
```

File must exist!

`is_open()`:
`true` if file could be opened
`false` in other case

Closing the file

Disconnects the variable from the text file

```
stream.close();
```

```
file.close();
```


Reading from Text Files

Performing read operations

- ✓ Extractor (>>) `file >> variable;`
  Skips heading blank spaces (space, tab, Enter, ...)
  Numerical data: reads until first invalid character is found
  Strings (`string`): reads until next blank space is found
- ✓ `file.get(c)`
  Reads next character in variable `c`, whichever it is
- ✓ `getline(file, string)`
  Reads in the *`string`* all the remaining characters in the line
  Including blank spaces
  Until the next new line is found (Enter is discarded)

With files the `sync()` function has no effect

Reading from Text Files

What am I going to read?

- ✓ A number
  Use the extractor `file >> num;`
- ✓ A character (whichever it is)
  Use the function `get()` `file.get(c);`
- ✓ A string without spaces
  Use the extractor `file >> str;`
- ✓ A string possibly with spaces
  Use the function `getline()` `getline(file, str);`

Luis Hernández Yáñez

Reading from Text Files

Where will the next input begin?

- ✓ After a number is read with the extractor:
  First invalid character (inc. blank spaces)

```
123.95 DVD Player
      ↑
```

- ✓ After a character is read with `get()`:
  Next character (inc. blank spaces)

```
123.95 DVD Player
 ↑
```

- ✓ After a string is read with the extractor:
  Next blank space (inc. Enter)

```
DVD Player
    ↑
```

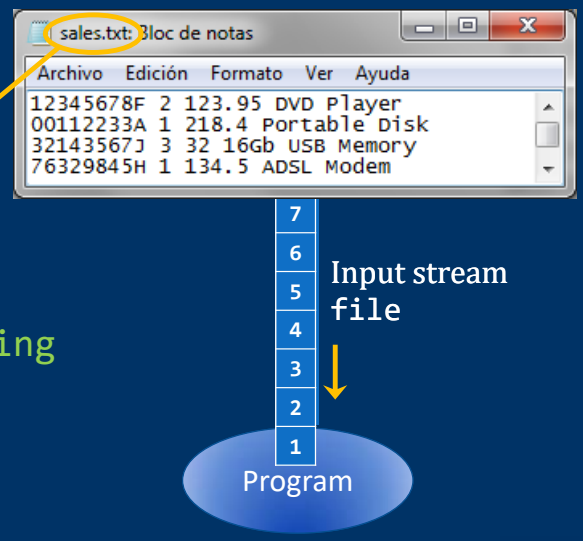
- ✓ After a string is read with `getline()`:
  Beginning of the next line

```
12345678F 2 123.95 DVD Player↲
00112233A 1 218.4 Portable Disk↲
↑
```

Luis Hernández Yáñez

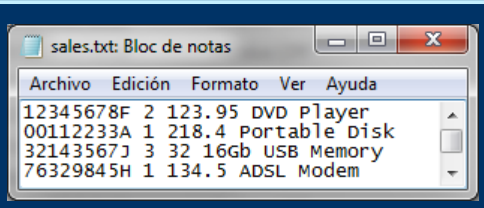
Reading from Text Files

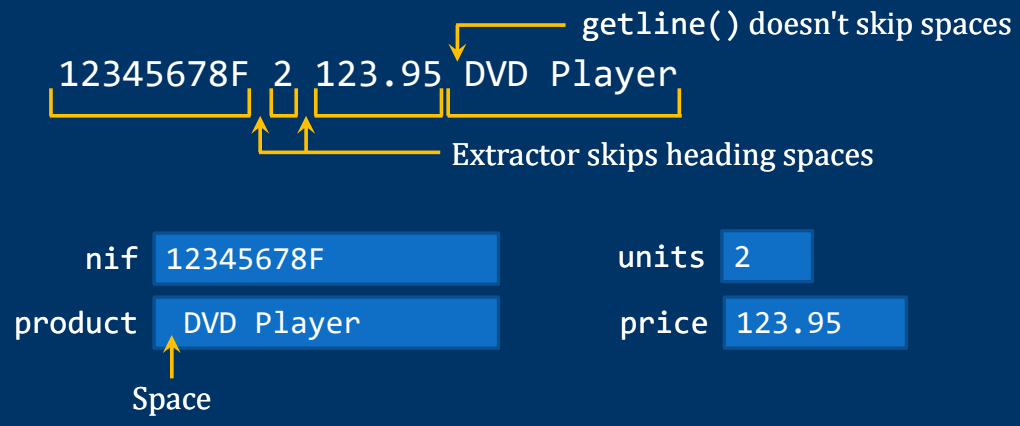
```
string nif, product;
int units;
double price;
char aux;

1 ifstream file;
2 file.open("sales.txt"); // Opening
3 file >> nif >> units >> price;
  getline(file, product);
4 file.close(); // Closing
```



Reading from Text Files

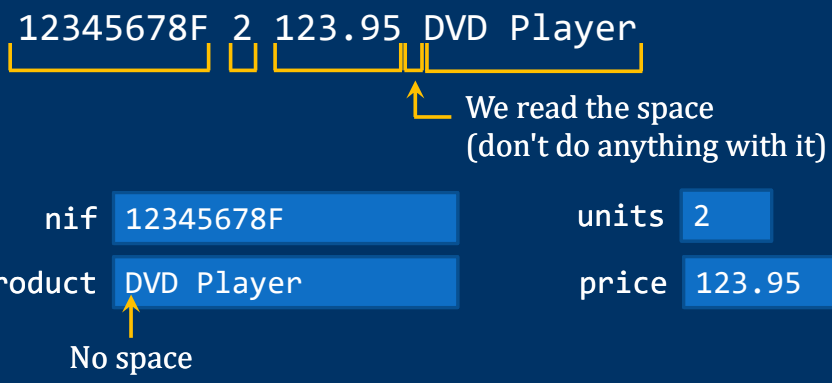
```
file >> nif;
file >> units;
file >> price;
getline(file, product);
```





Reading from Text Files

```
file >> nif;
file >> units;
file >> price;
file.get(aux); // Skips the space
getline(file, product);
```



Processing File Data

Each line: data of one sale

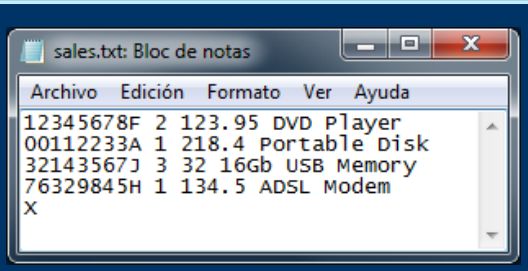
We want to print each sale total

units x price plus VAT (21%)

End of file data: "X" as NIF

Processing loop:

- ✓ In each cycle we process one line (sale)
- ✓ We can use the same variables in each cycle



Read first NIF

While NIF is different from X:

Read units, price and description

Calculate and print total

Read next NIF

Processing File Data

readfile.cpp

```
#include <iostream>
#include <string>
#include <fstream>
#include <iomanip> // Output format
using namespace std;

int main() {
   const int VAT = 21;
   string nif, product;
   int units;
   double price, net, total, vat;
   char aux;
   ifstream file;
   int counter = 0; // To count sales

   file.open("sales.txt");
```

.../...

Processing File Data

```
   if (file.is_open()) { // File exists
      file >> nif; // First NIF
      while (nif != "X") {
         file >> units >> price;
         file.get(aux); // Skips space
         getline(file, product);
         counter++;
         net = units * price;
         vat = net * VAT / 100;
         total = net + vat;
         cout << "Sale " << counter << ".-" << endl << "   "
            << product << ": " << units << " x " << fixed
            << setprecision(2) << price << " = " << net
            << " - VAT: " << vat << " - Total: " << total << endl;
         file >> nif; // Next NIF
      }
```

.../...

Processing File Data

```
        file.close();
    }
    else {
        cout << "ERROR: File couldn't be opened!" << endl;
    }

    return 0;
}
```

```
Sale 1.-
   DVD Player: 2 x 123.95 = 247.90 - VAT: 52.06 - Total: 299.96
Sale 2.-
   Portable Disk: 1 x 218.40 = 218.40 - VAT: 45.86 - Total: 264.26
Sale 3.-
   16Gb USB Memory: 3 x 32.00 = 96.00 - VAT: 20.16 - Total: 116.16
Sale 4.-
   ADSL Modem: 1 x 134.50 = 134.50 - VAT: 28.25 - Total: 162.75
```


Fundamentals of Programming I

Writing to Text Files

Writing to Text Files

Output Text Streams

`ofstream`

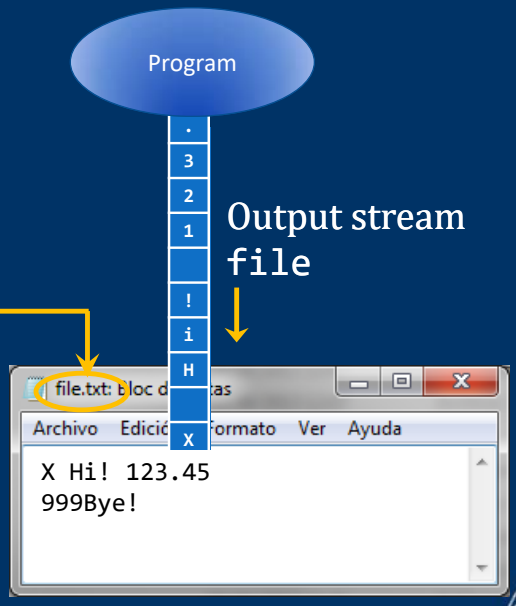
Creating a text file and writing to it:

1. Declare a variable of type `ofstream`
2. Associate the variable with the new text file (*create the file*)
3. Perform writing operations with << operator (insertor)
4. Dissociate the variable from the text file (*close the file*)

Remember!
If the file already exists all its contents will be erased
If the file is not closed information can be lost

Writing to Text Files

```
int value = 999;

1 ofstream file;
2 file.open("file.txt");
3 file << 'X' << " Hi! " << 123.45
     << endl << valor << "Bye!";
4 file.close();
```



Writing to Text Files

writefile.cpp

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

int main() {
   string nif, product;
   int units;
   double price;
   char aux;
   ofstream file;

   file.open("output.txt"); // Creation

   cout << "Client NIF (X to end): ";
   cin >> nif;
                                                    ...
```

Luis Hernández Yáñez

Writing to Text Files

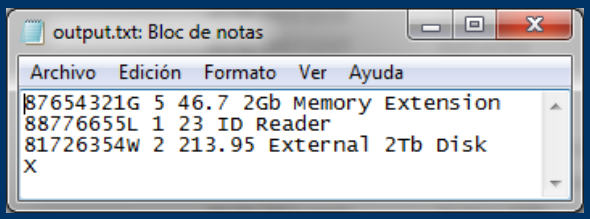
```
   while (nif != "X") {
      // Enter is pending from previous extraction...
      cin.get(aux); // Skips Enter
      cout << "Product: ";
      getline(cin, product);
      cout << "Units: ";
      cin >> units;
      cout << "Price: ";
      cin >> price;
      // Write data in one file's line...
      // With a space separating data
      file << nif << " " << units << " "
         << price << " " << product << endl;
      cout << "Client NIF (X to end): ";
      cin >> nif;
   }
                                                    ...
```

Luis Hernández Yáñez

Writing to Text Files

```
    // Write final sentinel...
    file << "X";
    file.close();

    return 0;
}
```

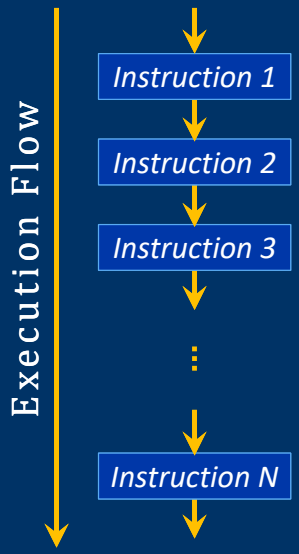
```
Client's NIF (X to end): 87654321G
Product: 2Gb Memory Extension
Units: 5
Price: 46.7
Client's NIF (X to end): 88776655L
Product: ID Reader
Units: 1
Price: 23
Client's NIF (X to end): 81726354W
Product: External 2Tb Disk
Units: 2
Price: 213.95
Client's NIF (X to end): X
```



Fundamentals of Programming I

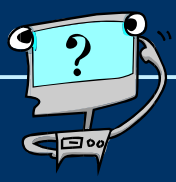
Execution Flow

Sequential Execution



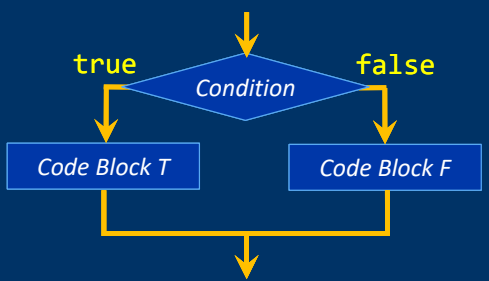

```
{
    double oper1, oper2, prod;
    cout << "Operand 1: ";
    cin >> oper1;
    cout << "Operand 2: ";
    ...
    cout << "Product: " << prod;
    return 0;
}
```

Selection



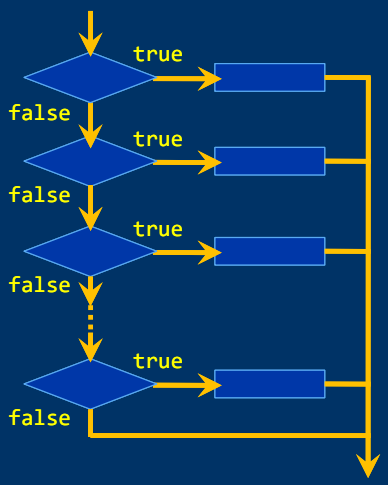
Choosing between two or more execution paths

Simple Selection (2 paths)



`if`

Multiple Selection (> 2 paths)



`if-else-if`

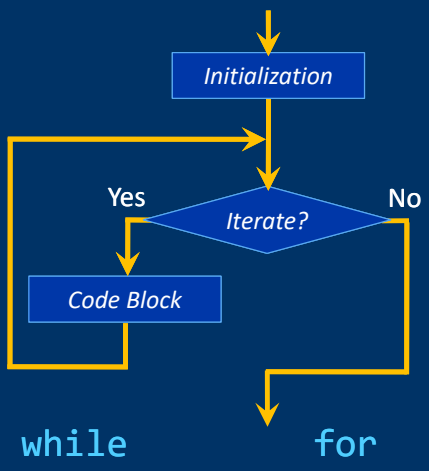
`switch`

Flow Diagrams

Repetition (Iteration)

Repeating execution of a group of instructions

Accumulation, collection processing, ...



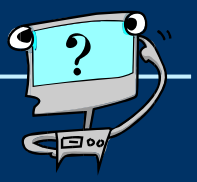
Luis Hernández Yáñez

Fundamentals of Programming I

Simple Selection

Luis Hernández Yáñez

Simple Selection (Bifurcation)

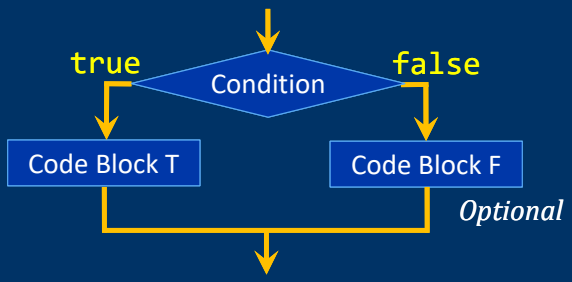


if instruction

```
if (condition) {
↦ Code T
}
[else {
↦ Code F
}]
```

condition: `bool` expression

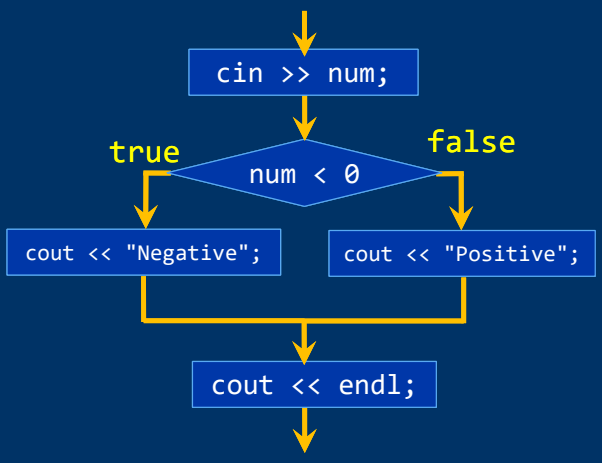
Optional `else` clause



`if` Instruction

sign.cpp

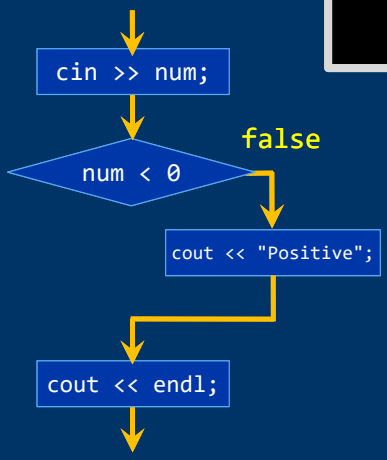
```
int num;
cin >> num;
if (num < 0) {
   cout << "Negative";
}
else {
   cout << "Positive";
}
cout << endl;
```



if Instruction

```
int num;
cin >> num;
if (num < 0) {
   cout << "Negative";
}
else {
   cout << "Positive";
}
cout << endl;
```

```
129
Positive
_
```

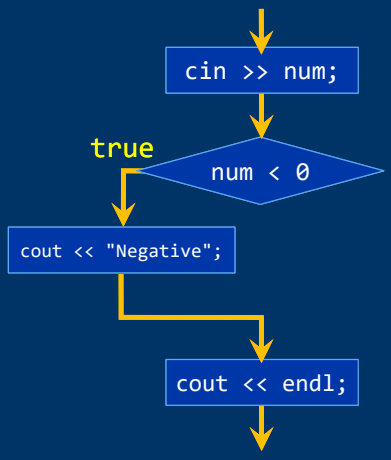


num 129

if Instruction

```
int num;
cin >> num;
if (num < 0) {
   cout << "Negative";
}
else {
   cout << "Positive";
}
cout << endl;
```

```
-5
Negative
_
```



num -5

Example

division.cpp

Division between two numbers protected against 0 denominator

```
double numerator, denominator, result;
cout << "Numerator: ";
cin >> numerator;
cout << "Denominator: ";
cin >> denominator;
if (denominator == 0)
   cout << "Impossible to divide by 0!";
else {
   result = numerator / denominator;
   cout << "Result: " << result << endl;
}
```


Fundamentals of Programming I

Logical Operators
(Compound Conditions)

Logical (Boolean) Operators

Apply to `bool` values (*conditions*)

Result: `bool`

!	NOT	Unary
&&	AND	Binary
||	OR	Binary

Operators (Priority)
. . .
!
* / %
+ -
< <= > >=
== !=
&&
||

Logical Operators - Truth Tables

!	
true	false
false	true

NOT

&&	true	false
true	true	false
false	false	false

AND

||	true	false
true	true	true
false	true	false

OR

```
bool cond1, cond2, result;
int a = 2, b = 3, c = 4;
result = !(a < 5);          // !(2 < 5) → !true → false
cond1 = (a * b + c) >= 12;  // 10 >= 12 → false
cond2 = (a * (b + c)) >= 12; // 14 >= 12 → true
result = cond1 && cond2;    // false && true → false
result = cond1 || cond2;    // false || true → true
```

Example

conditions.cpp

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

int main()
{
   int num;
   cout << "Enter a number between 1 and 10: ";
   cin >> num;
   if ((num >= 1) && (num <= 10))
      cout << "Number in valid value range";
   else
      cout << "Invalid number!";

   return 0;
}
```

Enclose simple conditions in parentheses!

Equivalent Conditions

```
((num >= 1) && (num <= 10))
 ((num > 0) && (num < 11))
((num >= 1) && (num < 11))
 ((num > 0) && (num <= 10))
```

Fundamentals of Programming I

Nesting ifs

Number of days in a month

monthdays.cpp

```
bool leapYear(int month, int year);

int main() {
   int month, year, days;
   cout << "Month number: ";
   cin >> month;
   cout << "Year: ";
   cin >> year;
   if (month == 2)
      if (leapYear(month, year))
         days = 29;
      else
         days = 28;
   else
      if ((month == 1) || (month == 3) || (month == 5) || (month == 7)
          || (month == 8) || (month == 10) || (month == 12))
         days = 31;
      else
         days = 30;
   ...
```


Leap Year?

Gregorian Calendar: Leap year if divisible by 4, except if century's last year (divisible by 100), unless also divisible by 400

```
bool leapYear(int month, int year) {
   bool isLeap;

   if ((year % 4) == 0) // Divisible by 4
      if (((year % 100) == 0) && ((year % 400) != 0))
         // But not last of century and not divisible by 400
         isLeap = false;
      else
         isLeap = true; // Leap year
   else
      isLeap = false;

   return isLeap;
}
```


`else` Clause Association

Each `else` belongs to previous unassociated `if` (same code block)

```
if (condition1) {
   if (condition2) {...}
   else {...}
}
else {
   if (condition3) {
      if (condition4) {...}
      if (condition5) {...}
      else {...}
   }
   else {...
```

Bad indentation can confuse...

```
if (x > 0) {
   if (y > 0) {...}
else {...}
```

```
if (x > 0) {
   if (y > 0) {...}
   else {...}
```

👀 Indentation helps to associate `else`s with `if`s

Fundamentals of Programming I

Conditions

Conditions

- Simple Condition:
  Logical expression (`true`/`false`) Without logical operators

```
num < 0
ch == 'a'
isalpha(ch)
12
```

Backward compatibility with C language:
`0` is equivalent to `false`
Every other value is equivalent to `true`

- Compound Condition:
  Simple conditions combined with logical operators

```
!isalpha(ch)
(num < 0) || (ch == 'a')
(num < 0) && ((ch == 'a') || !isalpha(ch))
```


Shortcut Boolean Evaluation

`true || X` ≡ `true`

```
(n == 0) || (x >= 1.0 / n)
```

If n is `0`: Division by zero? (Second condition)

If the first condition is `true`:

The second is not relevant and it won't be evaluated!

`false && X` ≡ `false`

```
(n != 0) && (x < 1.0 / n)
```

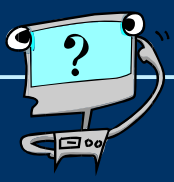
If n is `0`: Division by zero? (Second condition)

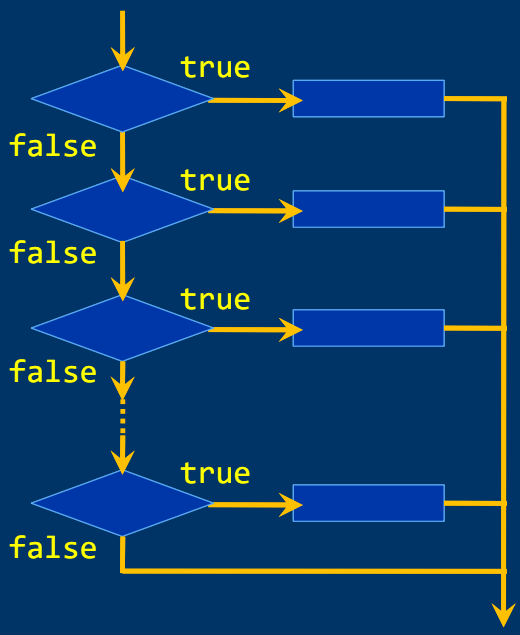
If the first condition is `false`:

The second is not relevant and it won't be evaluated!

Multiple Selection

Luis Hernández Yáñez






```
if-else-if
switch
```

Luis Hernández Yáñez

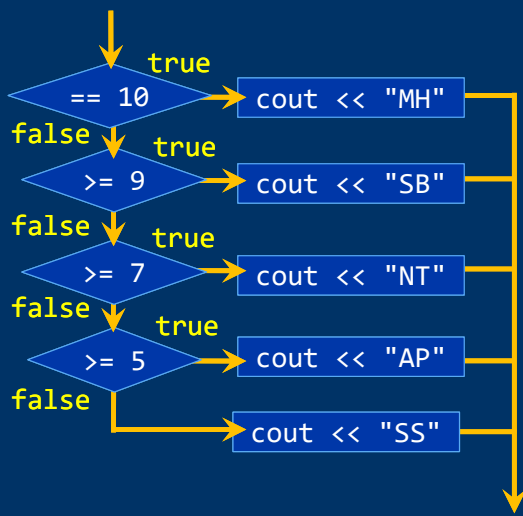
if-else-if Ladder

Luis Hernández Yáñez

if-else-if Ladder

Example:
Student letter grade based on his/her numerical grade (0-10)

If `grade == 10` then `MH`
else if `grade >= 9` then `SB`
else if `grade >= 7` then `NT`
else if `grade >= 5` then `AP`
else `SS`



Luis Hernández Yáñez

if-else-if Ladder

grade.cpp

```
double grade;
cin >> grade;
if (grade == 10)
   cout << "MH";
else
   if (grade >= 9)
      cout << "SB";
   else
      if (grade >= 7)
         cout << "NT";
      else
         if (grade >= 5)
            cout << "AP";
         else
            cout << "SS";
```

≡

```
double grade;
cin >> grade;
if (grade == 10)
   cout << "MH";
else if (grade >= 9)
   cout << "SB";
else if (grade >= 7)
   cout << "NT";
else if (grade >= 5)
   cout << "AP";
else
   cout << "SS";
```

if-else-if Ladder

Careful with condition order!

```
double grade;
cin >> grade;
if (grade < 5) { cout << "SS"; }
else if (grade < 7) { cout << "AP"; }
else if (grade < 9) { cout << "NT"; }
else if (grade < 10) { cout << "SB"; }
else { cout << "MH"; }
```

```
double grade;
cin >> grade;
if (grade >= 5) { cout << "AP"; }
else if (grade >= 7) { cout << "NT"; }
else if (grade >= 9) { cout << "SB"; }
else if (grade == 10) { cout << "MH"; }
else { cout << "SS"; }
```

Never executes!

Only shows AP or SS

if-else-if Ladder

Simplifying conditions


```
if (grade == 10) { cout << "MH"; }
else if ((grade < 10) && (grade >= 9)) { cout << "SB"; }
else if ((grade < 9) && (grade >= 7)) { cout << "NT"; }
else if ((grade < 7) && (grade >= 5)) { cout << "AP"; }
else if (grade < 5)  { cout << "SS"; }
```

Always `true`: `else` path
If not 10, less than 10
If not >= 9, less than 9
If not >= 7, less than 7
...
`true && X ≡ X`

```
if (grade == 10) { cout << "MH"; }
else if (grade >= 9) { cout << "SB"; }
else if (grade >= 7) { cout << "NT"; }
else if (grade >= 5) { cout << "AP"; }
else { cout << "SS"; }
```


Level Degree

`level.cpp`

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

int main() {
   int num;

   cout << "Enter level number: ";
   cin >> num;
   if (num == 4)
      cout << "Very High" << end;
   else if (num == 3)
      cout << "High" << endl;
   else if (num == 2)
      cout << "Medium" << endl;
   else if (num == 1) {
      cout << "Low" << endl;
   else {
      cout << "Invalid value!" << endl;

   return 0;
}
```

If num == 4 then Very High
If num == 3 then High
If num == 2 then Medium
If num == 1 then Low

Repeated Code in all Branches?

```
if (num == 4) cout << "Very High" << endl;
else if (num == 3) cout << "High" << endl;
else if (num == 2) cout << "Medium" << endl;
else if (num == 1) cout << "Low" << endl;
else cout << "Invalid value!" << endl;
```

```
if (num == 4) cout << "Very High";
else if (num == 3) cout << "High";
else if (num == 2) cout << "Medium";
else if (num == 1) cout << "Low";
else cout << "Invalid value!";
cout << endl;
```

Luis Hernández Yáñez

Fundamentals of Programming I

switch Instruction

Luis Hernández Yáñez

switch Instruction

Selection between the possible values of an expression

```
switch (expression) {
case constant1:
   {
      code1
   }
   [break;]
case constant2:
   {
      code2
   }
   [break;]
...
case constantN:
   {
      codeN
   }
   [break;]
[default:
   {
      defaultCode
   }]
}
```

{ and } can be omitted in cases

switch Instruction

level2.cpp

```
switch (num) {
case 4:
   cout << "Very High";
   break;
case 3:
   cout << "High";
   break;
case 2:
   cout << "Medium";
   break;
case 1:
   cout << "Low";
   break;
default:
   cout << "Invalid value!";
}
```

If num == 4 → Very High
If num == 3 → High
If num == 2 → Medium
If num == 1 → Low

break Instruction

Interrupts the `switch`; continues in the following instruction

```
switch (num) {
...
case 3:
   cout << "High";
   break;
case 2:
   cout << "Medium";
   break;
...
}
```

```
Number: 3
High
```

Luis Hernández Yáñez

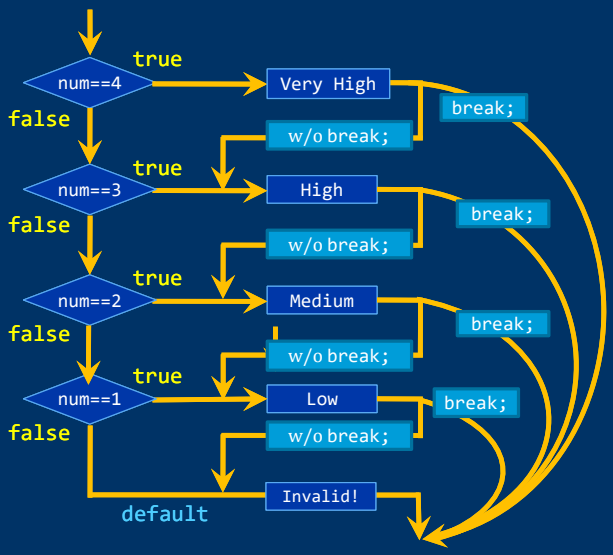
break Instruction

```
switch (num) {
...
case 3:
   cout << "High";
case 2:
   cout << "Medium";
case 1:
   cout << "Low";
default:
   cout << "Invalid number!";
}
```

```
Number: 3
HighMediumLowInvalid number!
```

Luis Hernández Yáñez

With and w/o break



A Menu

```
int menu() {
   int op = -1; // Any non-valid option

   while ((op < 0) || (op > 4)) {
      cout << "1 - New Client" << endl;
      cout << "2 - Edit Client" << endl;
      cout << "3 - Delete Client" << endl;
      cout << "4 - Show Client" << endl;
      cout << "0 - Exit" << endl;
      cout << "Option: ";
      cin >> op;

      if ((op < 0) || (op > 4))
         cout << "Invalid option! Try again" << endl;
   }

   return op;
}
```

```
1 - New Client
2 - Edit Client
3 - Delete Client
4 - Show Client
0 - Exit
Option: 5
Invalid option! Try again
1 - New Client
2 - Edit Client
3 - Delete Client
4 - Show Client
0 - Exit
Option: 3
```

Using the menu

```
int option;
...
option = menu();
switch (option) {
case 1:
   cout << "In Option 1..." << endl;
   break;
case 2:
   cout << "In Option 2..." << endl;
   break;
case 3:
   cout << "In Option 3..." << endl;
   break;
case 4:
   cout << "In Option 4..." << endl;
} // In the last case we don't need a break instruction
```


A Loop for the Menu

```
int option;
...
option = menu();
while (option != 0) {
   switch (option) {
   case 1:
      cout << "In Option 1..." << endl;
      break;
   ...
   case 4:
      cout << "In Option 4..." << endl;
   } // switch
   ...
   option = menu();
} // while
```


Multiple Cases

grade2.cpp

```
int grade; // No decimals
cout << "Grade (0-10): ";
cin >> grade;
switch (grade) {
case  0:
case  1:
case  2:
case  3:
case  4:
   cout << "Suspenso";
   break; // From 0 to 4: SS
case  5:
case  6:
   cout << "Aprobado";
   break; // 5 or 6: AP
case  7:
case  8:
   cout << "Notable";
   break; // 7 or 8: NT
case  9:
case 10:
   cout << "Sobresaliente";
   break; // 9 or 10: SB
default:
   cout << "Invalid!";
}
```

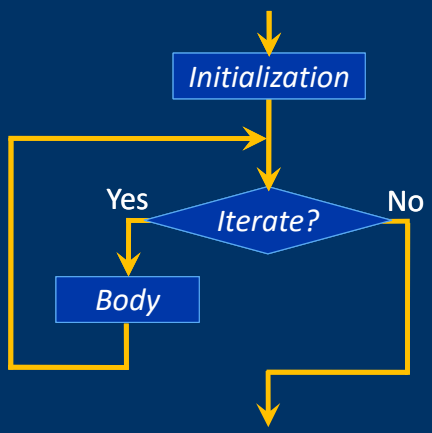

Printing Enumerated Types Values

```
typedef enum { January, February, March, April, May, June, July,
               August, September, October, November, December } tMonth;
tMonth month;
...
switch (month) {
case January:
   cout << "January";
   break;
case February:
   cout << "February";
   break;
...
case December:
    cout << "December";
}
```


Repetition

Luis Hernández Yáñez

Repetition (Iteration)



`while` and `for` Loops

Luis Hernández Yáñez

Loops

- ✓ Variable number of iterations:
  - `while` loop

    `while (condition) body`

    Executes the *body* (code block) while the *condition* is `true`
  - `do-while` loop

    Checks the condition at the end (Lesson 5)
- ✓ Fixed number of iterations:
  - `for` loop

    `for (initialization; condition; step) body`

    Executes the *body* (code block) while the *condition* is `true`

    A counter (integer variable) is used and initialized

Fundamentals of Programming I

`while` Loop

while Loop

while.cpp

While condition is true, executes the body

```
while (condition) {
   body
}
```

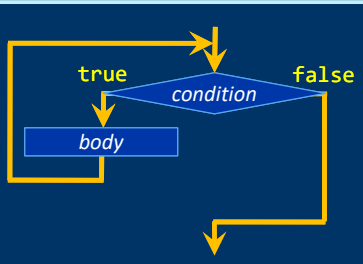
Condition at the beginning

```
int i = 1; // Initialization of variable i
while (i <= 100) {
   cout << i << endl;
   i++;
}
```

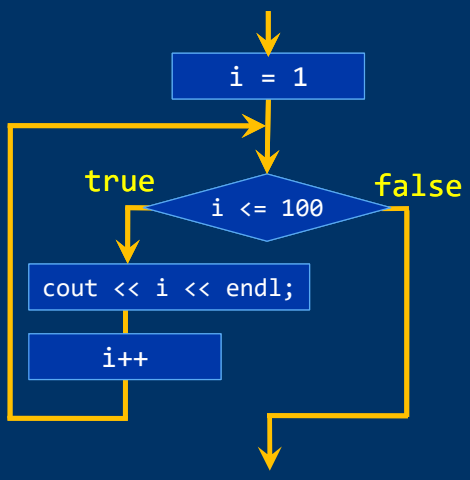
Prints integer numbers from 1 to 100

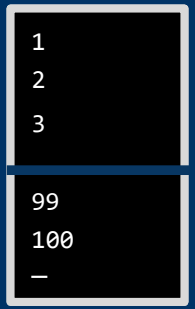
while Loop Execution

```
int i = 1;
while (i <= 100) {
   cout << i << endl;
   i++;
}
```



i 101





while Loop

What if the condition is false at the beginning?

The loop's body is never executed

```
int op;
cout << "Enter your option: ";
cin >> op;
while ((op < 0) || (op > 4)) {
   cout << "Invalid! Try again" << endl;
   cout << "Enter your option: ";
   cin >> op;
}
```

If the user enters a number between 0 and 4:

The loop's body is never executed

while example

first.cpp

First integer whose square is higher than 1000

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

int main() {
   int num = 1;

   while (num * num <= 1000)
      num++;

   cout << "First integer with square higher than 1,000: "
      << num << endl;

   return 0;
}
```

Execute the program to find out which number it is!

← We start with 1

← Incremented by 1

Sequence of numbers 1, 2, 3, 4, 5, ...

Sum and mean of numbers

summean.cpp

```
#include <iostream>
using namespace std;
int main() {
   double num, sum = 0, mean = 0;
   int count = 0;
   cout << "Enter a number (0 to end): ";
   cin >> num;
   while (num != 0) { // 0 ends loop
      sum = sum + num;
      count++;
      cout << "Enter a number (0 to end): ";
      cin >> num;
   }
   if (count > 0)
      mean = sum / count;
   cout << "Sum = " << sum << endl << "Mean = " << mean << endl;
   return 0;
}
```

Process the *sequence* of numbers entered

← First one is read

← Next one is read

Fundamentals of Programming I

for Loop

for Loop

Fixed number of iterations

Counter variable that determines the number of iterations:

```
for ([int] var = ini; condition; step) body
```

The *condition* compares *var*'s value with a final value

The *step* increases or decreases *var*'s value

var's value must converge to the final value

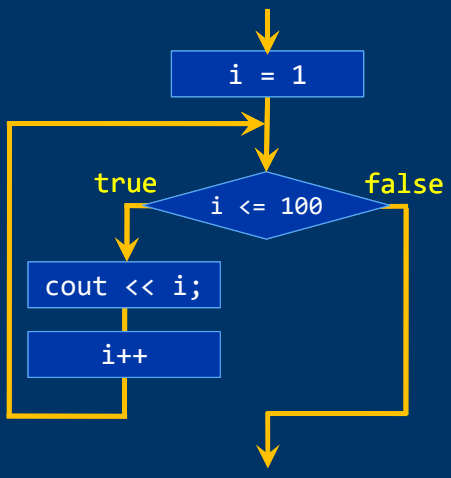
```
for (int i = 1; i <= 100; i++)...      1, 2, 3, 4, 5, ..., 100
for (int i = 100; i >= 1; i--)...      100, 99, 98, 97, ..., 1
```

As many iterations as values that the counter variable gets

for Loop Execution

```
for (initialization; condition; step) body
```

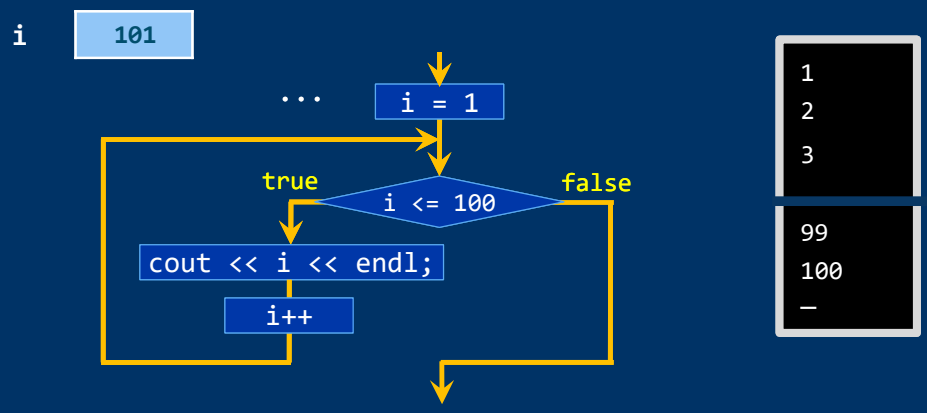
```
for (int i = 1; i <= 100; i++) {
    cout << i;
}
```



for Loop Execution

for1.cpp

```
for (int i = 1; i <= 100; i++) {
    cout << i << endl;
}
```



for Loop

for2.cpp

Counter variable

The *step* can increase/decrease by more than one:

```
for (int i = 1; i <= 100; i = i + 2)
    cout << i << endl;
```

This `for` loop prints odd numbers from 1 to 99

Very Important!
The loop's body must NEVER alter the counter's value

Termination guarantee

All loops must finish execution sooner or later

`for` loops: counter variable must converge to the final value

for Loop Example

sum.cpp

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

long long int sum(int n);

int main() {
   int num;
   cout << "Final number: ";
   cin >> num;
   if (num > 0) // The number must be positive
      cout << "Sum of the numbers from 1 to " << num << ": " << sum(num);
   return 0;
}

long long int sum(int n) {
   long long int total = 0;
   for (int i = 1; i <= n; i++)
      total = total + i;
   return total;
}
```

$$\sum_{i=1}^{N} i$$

Process the *sequence* of numbers
1, 2, 3, 4, 5, ..., *n*

for Loop

Increment/decrement operator prefix or postfix?

It doesn't matter

This two loops produce the same result:

```
for (int i = 1; i <= 100; i++) ...
for (int i = 1; i <= 100; ++i) ...
```

Infinite loops

```
for (int i = 1; i <= 100; i--) ...
     1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 ...
```

Further and further from the final value (100)

A design/programming error

Counter Variable Scope

Declared in the `for` *instruction*

```
for (int i = 1; ...)
```

Only known in loop's body (its scope)

We can't use it in instructions following the loop

Declared before the loop

```
int i;
for (i = 1; ...)
```

It's known in loop's body and after the loop

Scope external to the loop

`for` Loop *versus* `while` Loop

`for` loops can be rewritten as conditional loops

```
for (int i = 1; i <= 100; i++) body
```

Is equivalent to:

```
int i = 1;
while (i <= 100) {
   body
   i++;
}
```

The reverse is not always possible:

Equivalent `for` *loop?*

We don't know how many numbers the user will enter!

```
int i;
cin >> i;
while (i != 0) {
   body
   cin >> i;
}
```


Nested Loops

Luis Hernández Yáñez

Nested `for` Loops

A `for` loop in the body of another `for` loop

Each one with its own counter variable:

```
for (int i = 1; i <= 100; i++) {
    for (int j = 1; j <= 5; j++) {
        body
    }
}
```

For each `i` value `j` gets the values from `1` to `5`

`j` *varies quicker than* `i`

i	j
1	1
1	2
1	3
1	4
1	5
2	1
2	2
2	3
2	4
2	5
3	1
...	

Luis Hernández Yáñez

BY NC SA

Multiplication Tables

tables.cpp

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

int main() {
   for (int i = 1; i <= 10; i++)
      for (int j = 1; j <= 10; j++)
         cout << setw(2) << i << " x "
              << setw(2) << j << " = "
              << setw(3) << i * j << endl;

   return 0;
}
```

```
 1 x  1 =   1
 1 x  2 =   2
 1 x  3 =   3
 1 x  4 =   4
 ...
 1 x 10 =  10
 2 x  1 =   2
 2 x  2 =   4
 ...
10 x  7 =  70
10 x  8 =  80
10 x  9 =  90
10 x 10 = 100
```

A Better Presentation

tables2.cpp

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

int main() {
   for (int i = 1; i <= 10; i++) {
      cout << "Table of " << i << endl;
      cout << "--------------" << endl;
      for (int j = 1; j <= 10; j++)
         cout << setw(2) << i << " x "
              << setw(2) << j << " = "
              << setw(3) << i * j << endl;
      cout << endl;
   }

   return 0;
}
```

```
D:\FP\Less03>tables2
Table of 1
--------------
 1 x  1 =   1
 1 x  2 =   2
 1 x  3 =   3
 1 x  4 =   4
 1 x  5 =   5
 1 x  6 =   6
 1 x  7 =   7
 1 x  8 =   8
 1 x  9 =   9
 1 x 10 =  10

Table of 2
--------------
 2 x  1 =   2
 2 x  2 =   4
 2 x  3 =   6
 2 x  4 =   8
 2 x  5 =  10
 2 x  6 =  12
 2 x  7 =  14
 2 x  8 =  16
```

Nested Loops

menu.cpp

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

int menu(); // 1: Multiplication tables; 2: Summation
long long int sum(int n); // Summation

int main() {
   int option = menu();
   while (option != 0) {
      switch (option) {
      case 1:
         for (int i = 1; i <= 10; i++)
            for (int j = 1; j <= 10; j++)
               cout << setw(2) << i << " x " << setw(2) << j
                    << " = " << setw(3) << i * j << endl;
         break;
```


Nested Loops

```
      case 2:
         int num = 0;
         while (num <= 0) {
            cout << "Final number (positive)? ";
            cin >> num;
         }
         cout << "Sum of the numbers from 1 to "
              << num << ": " << sum(num) << endl;
         break;
      } // switch
      option = menu();
   } // while (option != 0)

   return 0;
}
```


Nested Loops

```
int menu() {
   int op = -1;
   while ((op < 0) || (op > 2)) {
      cout << "1 - Multiplication Tables" << endl;
      cout << "2 - Summation" << endl;
      cout << "0 - Exit" << endl;
      cout << "Option: ";
      cin >> op;
      if ((op < 0) || (op > 2))
         cout << "Invalid option! Try again" << endl;
   }
   return op;
}

long long int sum(int n) {
   long long int total = 0;
   for (int i = 1; i <= n; i++)
      total = total + i;
   return total;
}
```


Both Kind of Loops Nested

```
while (option != 0) {
   ...
      for (int i = 1; i <= 10; i++) {
         for (int j = 1; j <= 10; j++) {
            ...
         }
      } ...
      while (num <= 0) {
         ...
      } ...                          sum()
      for (int i = 1; i <= n; i++) {
         ...
      } ...
   while ((op < 0) || (op > 2)) {
      ...
   } ...                             menu()
}
```


Scope and Visibility

Luis Hernández Yáñez

Identifier Scope

Each code block creates a new scope:

```
int main() {
  double d = -1, sum = 0;
  int count = 0;
  while (d != 0) {
    cin >> d;
    if (d != 0) {
      sum = sum + d;
      count++;
    }
  }
  cout << "Sum = " << sum << endl;
  cout << "Mean = " << sum / count << endl;
  return 0;
}
```

3 nested scopes

Luis Hernández Yáñez

Identifier Scope

An identifier is known
in the scope it has been declared in,
from the line where the declaration instruction is,
and in the following subscopes

Luis Hernández Yáñez

Identifier Scope

variable d scope

```
int main() {
   double d;
   if (...) {
      int count = 0;
      for (int i = 0; i <= 10; i++) {
         ...
      }
   }
   char c;
   if (...) {
      double x;
      ...
   }
   return 0;
}
```

Luis Hernández Yáñez

Identifier Scope

```
int main() {
   double d;
   if (...) {
      int count = 0;
      for (int i = 0; i <= 10; i++) {
         ...
      }
   }
   char c;
   if (...) {
      double x;
      ...
   }
   return 0;
}
```

variable count scope

Identifier Scope

```
int main() {
   double d;
   if (...) {
      int count = 0;
      for (int i = 0; i <= 10; i++) {
         ...
      }
   }
   char c;
   if (...) {
      double x;
      ...
   }
   return 0;
}
```

variable i scope

Identifier Scope

```
int main() {
   double d;
   if (...) {
      int count = 0;
      for (int i = 0; i <= 10; i++) {
         ...
      }
   }
   char c;
   if (...) {
      double x;
      ...
   }
   return 0;
}
```

variable c scope

Identifier Scope

```
int main() {
   double d;
   if (...) {
      int count = 0;
      for (int i = 0; i <= 10; i++) {
         ...
      }
   }
   char c;
   if (...) {
      double x;
      ...
   }
   return 0;
}
```

variable x scope

Identifier Visibility

If in one subscope an identifier is declared
with the identical name
as one declared in the scope,
the one in the subscope *hides* the one in the scope

Luis Hernández Yáñez

Identifier Visibility

```
int main() {
   int i, x;
   if (...) {
      int i = 0;
      for(int i = 0; i <= 10; i++) {
         ...
      }
   }
   char c;
   if (...) {
      double x;
      ...
   }
   return 0;
}
```

Hides previous `i` in this scope

Hides previous `i` in this scope

Hides previous `x` in this scope

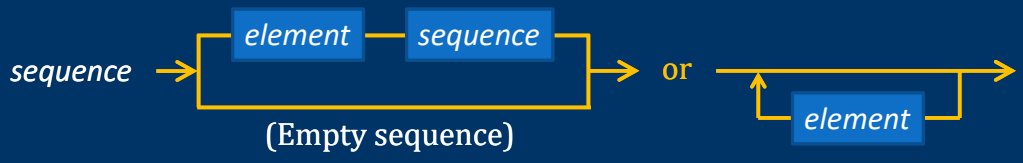
Luis Hernández Yáñez

Sequences

Luis Hernández Yáñez

Sequences

Succession of elements of the same type with linear access



1 34 12 26 4 87 184 52

It starts with a *first* element (if not empty)

Every element is followed by a sequence (empty, if it is the *last* one)

Sequential (linear) access

We always start by accessing the first element

We can only access the next element (*successor*), if there is one

All the elements of the same type

Luis Hernández Yáñez

Sequences in Programming

No infinite sequences: There's always a final element!

- ✓ Explicit Sequences:
  - Data succession from a device (keyboard, disk, sensor, ...)
- ✓ Calculated Sequences:
  - Recurrence formula that determines the next element
- ✓ Lists (elements stored in memory)

Explicit sequences we will manage:

Data entered with the keyboard or read from a file

With a special final element (*sentinel*)

```
1  34  12  26  4  87  184  52  -1
```

Detecting the End of the Sequence

- ✓ Explicit sequence read from file:
  - Detect end-of-file mark (`Eof`)
  - Detect a sentinel value at the end ←
- ✓ Explicit sequence read from keyboard:
  - Ask the user if he/she wants to enter more data
  - Ask the user first how many data he/she is going to enter
  - Detect a sentinel value at the end ←

Sentinel value: Special value at the end that can't be in the actual sequence

For example: Positive number sequence → sentinel: any negative number

```
12  4  37  23  8  19  83  63  2  35  17  76  15  -1
                                                  ↓
```

Sentinels

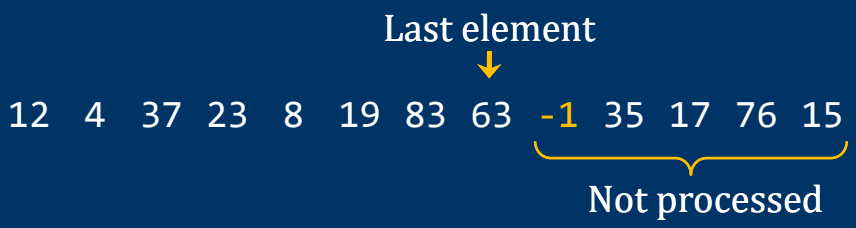
There must exist at least one non-valid value

Numerical sequences: If every number is valid, there is no possible sentinel

Character strings: Control (non-printable) characters?

Actually the sentinel value is part of the sequence,
but its meaning is special and it is not processed like the others

It means that we have reached the end of the sequence
(*even if there are more values after the sentinel*)



Sequence Processing Schemes

Processing sequence elements one by one from the first one

Traversal

Same processing for all sequence elements

To show all the elements in a sequence, to sum all the numbers in a sequence, to know which ones are even numbers, ...

Traversal ends when the end of the sequence is reached

Search

Traversal of the sequence searching for a specific element

To locate the first number in a sequence greater than 1,000

Search ends when the element searched for is found
or when the end of the sequence is reached (*not found*)

Sequence Traversal

Luis Hernández Yáñez

Traversal Scheme

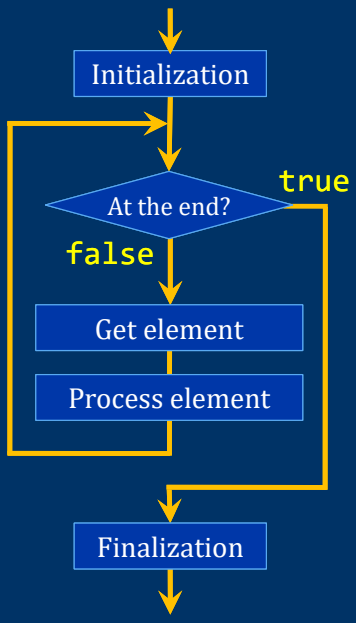
Same processing for all the sequence elements

Initialization
While not at the end of the sequence:
Get next element
Process element
Finalization

At the beginning the first element of the sequence is processed
In the next loop steps the next elements are processed

Luis Hernández Yáñez

Traversal Scheme



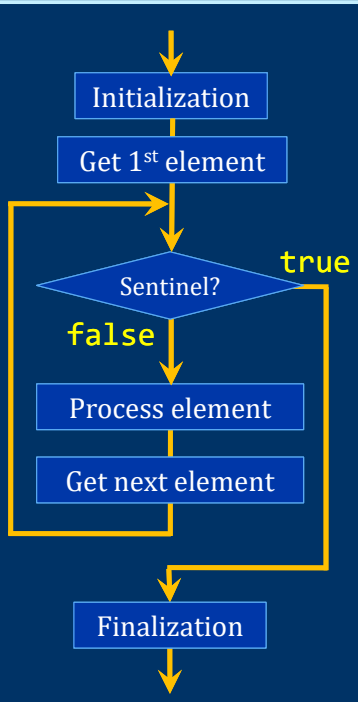
We don't know how many elements there are
→ Can't be implemented with a `for` loop

Luis Hernández Yáñez

Explicit Sequences with Sentinel

Implementation with `while`

Initialization
Get first element
While not the sentinel:
    Process element
    Get next element
Finalization



Luis Hernández Yáñez

Explicit Sequences Read from the Keyboard

Positive number sequence

At least one reading, sentinel = -1

```
double d, sum = 0;                          Initialization
cout << "Value (-1 ends): ";
cin >> d;                                   First element
while (d != -1) {                           While not the sentinel
   sum = sum + d;                           Process element
   cout << "Value (-1 ends): ";
   cin >> d;                                Next element
}
cout << "Sum = " << sum << endl;            Finalization
```


Explicit Sequences Read from the Keyboard

Length of a character sequence

length.cpp

Sentinel: dot character (.)

```
int length() {
   int l = 0;
   char c;
   cout << "Text terminated with a dot: ";
   cin >> c;                  // Get first character
   while (c != '.') {         // While not the sentinel
      l++;                    // Process
      cin >> c;               // Get next element
   }
   return l;
}
```


Explicit Sequences Read from the Keyboard

count.cpp

How many times does a character appear in a string?

Sentinel: asterisk (*)

```
char searched, c;
int count = 0;
cout << "Character to count: ";
cin >> searched;
cout << "String: ";
cin >> c;                          — Get first element
while (c != '*') {                 — While not the sentinel
   if (c == searched)              }
      count++;                     } Process element
   cin >> c;                       — Get next element
}
cout << searched << " appears " << count << " times";
```

Explicit Sequences Read from a File

sum2.cpp

Sum of the numbers in a sequence

Sentinel: 0

```
double sumSequence() {
   double d, sum = 0;
   ifstream file; // Input file
   file.open("data.txt");
   if (file.is_open()) {
      file >> d;              // Get first number
      while (d != 0) {        // While not the sentinel
         sum = sum + d;       // Process the number
         file >> d;           // Get next number
      }
      file.close();
   }
   return sum;
}
```

Calculated Sequences

Luis Hernández Yáñez

Calculated Sequences

summation.cpp

Recurrence: $e_{i+1} = e_i + 1 \qquad e_1 = 1$

1 2 3 4 5 6 7 8 ...

$$\sum_{i=1}^{N} i$$

Sum of the numbers in the calculated sequence:

```
int main() {
   int num;
   cout << "N = ";
   cin >> num;
   cout << "Summation:" << sum(num);
   return 0;
}
long long int sum(int n) {
   long long int summation = 0;
   for (int i = 1; i <= n; i++)
      summation = summation + i;
   return summation;
}
```

Last element in the sequence: n

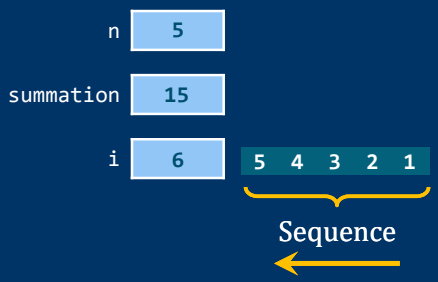
Luis Hernández Yáñez

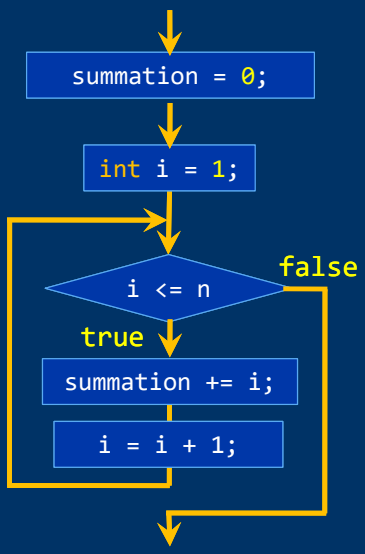
BY NC SA

Sum of the Numbers in a Calculated Sequence

```
long long int sum(int n) {
   long long int summation = 0;
   for (int i = 1; i <= n; i++)
      summation = summation + i;
   ...
```

$$\sum_{i=1}^{N} i$$





Fibonacci Numbers

Definition

$F_i = F_{i-1} + F_{i-2}$

$F_1 = 0$

$F_2 = 1$

0 1 1 2 3 5 8 13 21 34 55 89 ...

End of the sequence?

First number greater than another given number

That number acts as a sentinel

If num is 50, then the sequence will be:

0 1 1 2 3 5 8 13 21 34

Fibonacci Numbers

fibonacci.cpp

Traversal of the calculated sequence

```
int num, fib, fibMinus2 = 0, fibMinus1 = 1; // 1st and 2nd
fib = fibMinus2 + fibMinus1; // Calculate the third
cout << "To: ";
cin >> num;
if (num >= 1) { // Must be a positive integer
   cout << "0 1 "; // First and second are <= num
   while (fib <= num) { // While not greater than num
      cout << fib << " ";
      fibMinus2 = fibMinus1; // Update the predecessors
      fibMinus1 = fib; // to obtain...
      fib = fibMinus2 + fibMinus1; // ... the next one
   }
}
```

Too many comments? Maybe an explanation before the code fragment is better

Fibonacci Numbers

The loop generates the elements in the sequence:

```
while (fib <= num) {
   cout << fib << " ";
   fibMinus2 = fibMinus1;
   fibMinus1 = fib;
   fib = fibMinus2 + fibMinus1;
}
```

num	100
fib	5
fibMinus1	3
fibMinus2	2

0 1 1 2 3 5...

Searching in Sequences

Luis Hernández Yáñez

Searching Scheme

Locating the first element with a certain property

Initialization

While the element is not found and not at the end of the sequence:

Get next element

Test if the element satisfies the condition

Finalization (process the element or indicate that it was not found)

Searched element: must meet certain criteria

Two conditions for the loop to end: element found or at the end

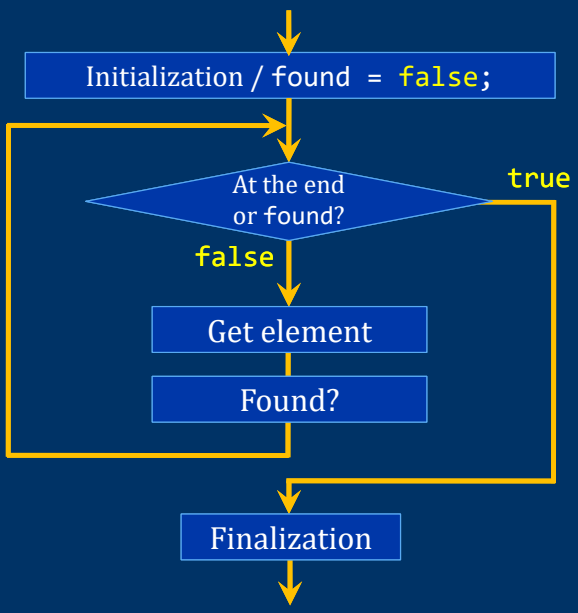
A logical variable to indicate if it has been found

Luis Hernández Yáñez

BY NC SA

Searching Scheme

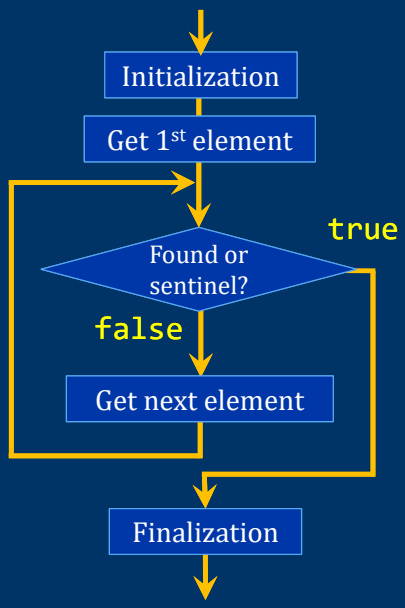
Locating the first element with a certain property



Explicit Sequences with Sentinel

Implementation with `while`

Initialization
Get the first element
While not found nor the sentinel:
    Get the next element
Finalization (found?)



Explicit Sequences Read from the Keyboard

First element greater than a given number

search.cpp

Sentinel: -1

```
double d, num;
bool found = false;
cout << "Find the first element greater than: ";
cin >> num;
cout << "Next element (-1 to end): ";
cin >> d; // Get the first element
while ((d != -1) && !found)
// While not the sentinel and not found
   if (d > num) // Found?
      found = true;
   else {
      cout << "Next element (-1 to end): ";
      cin >> d; // Get the next element
   }
```


Fundamentals of Programming I

Arrays of Simple Data Types

Arrays

Homogeneous Collections

Same data type for a collection of data:

Grades for the students in a class, sale for every weekday, temperature for every day in the month, ...

Instead of declaring *N* variables...

sMon	sTue	sWed	sThu	sFri	sSat	sSun
125.40	76.95	328.80	254.62	435.00	164.29	0.00

... we declare a table of *N* values:

sales	125.40	76.95	328.80	254.62	435.00	164.29	0.00
Indexes →	0	1	2	3	4	5	6

Arrays

Sequential Structure

Each element in a specific position (*index*):

- ✓ Indexes are positive integers
- ✓ The index of the first element is always 0
- ✓ Indexes are incremented by one

sales	125.40	76.95	328.80	254.62	435.00	164.29	0.00
	0	1	2	3	4	5	6

Direct Access

Each element is accessed by its index with the operator `[]`:

`sales[4]` accesses the 5th element (it contains the value `435.00`)

```
cout << sales[4];
sales[4] = 442.75;
```

Data of the same base type: Used as any other variable

Array Types

Array Type Declaration

```
typedef base_type type_name[length];
```

Examples:

```
typedef double tTemp[7];
typedef short int tMonthDays[12];
typedef char tVowels[5];
typedef double tSales[31];
typedef tCoin tChange[15]; // Enumerated base type
```

Remember: We create type names by starting with a lowercase `t` followed by one or more words with their first letter capitalized

Array Variables

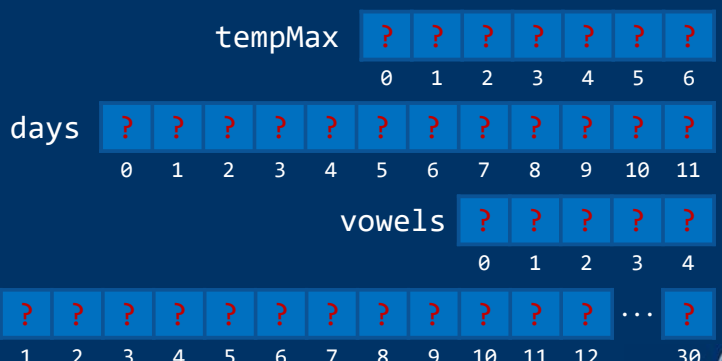
Array Variable Declaration

```
type_name variable_name;
```

Examples:

```
tTemp tempMax;
tMonthDays days;
tVowels vowels;
tSales maySales;
```

```
typedef double tTemp[7];
typedef short int tMonthDays[12];
typedef char tVowels[5];
typedef double tSales[31];
```



Elements are NOT automatically initialized

Array Variable Use

Luis Hernández Yáñez

Array Element Access

```
typedef char tVowels[5];
```

name[*index*]

Each element is accessed by its index (position in the array)

```
tVowels vowels;
```

vowels

'a'	'e'	'i'	'o'	'u'
0	1	2	3	4

5 elements, indexes from 0 to 4:

`vowels[0]` `vowels[1]` `vowels[2]` `vowels[3]` `vowels[4]`

Processing each element:

As any other variable of that base type

```
cout << vowels[4];
vowels[3] = 'o';
if (vowels[i] == 'e') ...
```

Luis Hernández Yáñez

CC BY NC SA

Array Element Access

IMPORTANT!

The compiler doesn't test if the index is valid!

It is the programmer's responsibility!

```
const int Dim = 100;
typedef double tSales[Dim];
tSales sales;
```

👀 Declare the length of the arrays as a constants

Valid indexes: integers from 0 to Dim-1

`sales[0]` `sales[1]` `sales[2]` ... `sales[98]` `sales[99]`

What is `sales[100]`? Or `sales[-1]`? Or `sales[132]`?

Memory of another program's variable!

Fundamentals of Programming I

Array Traversal

Array Traversal

Arrays: Fixed length → Fixed number of iterations loop (for)

Example: Mean of a sequence of temperature measures

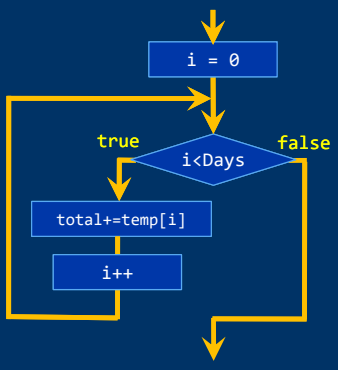
```
const int Days = 7;
typedef double tTemp[Days];
tTemp temp;
double mean, total = 0;
...
for (int i = 0; i < Days; i++)
   total = total + temp[i];
mean = total / Days;
```

Luis Hernández Yáñez

Array Traversal

12.40	10.96	8.43	11.65	13.70	13.41	14.07
0	1	2	3	4	5	6

```
tTemp temp;
double mean, total = 0;
...
for (int i = 0; i < Days; i++)
   total = total + temp[i];
```



Memoria

Days	7
temp[0]	12.40
temp[1]	10.96
temp[2]	8.43
temp[3]	11.65
temp[4]	13.70
temp[5]	13.41
temp[6]	14.07
mean	?
total	84.62
i	7

Luis Hernández Yáñez

Array Traversal

tempmean.cpp

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

const int Days = 7;
typedef double tTemp[Days];

double mean(const tTemp temp);

int main() {
   tTemp temp;
   for (int i = 0; i < Days; i++) { // Array traversal
      cout << "Temperature of the day " << i + 1 << ": ";
      cin >> temp[i];
   }
   cout << "Mean temperature: " << mean(temp) << endl;
   return 0;
}
```

Users use 1 to 7 to enumerate days
Program interface must be user-friendly, although internally indexes from 0 to 6 are used

.../...

Array Traversal

```
double mean(const tTemp temp) {
   double meanValue, total = 0;

   for (int i = 0; i < Days; i++) // Array traversal
      total = total + temp[i];
   meanValue = total / Days;

   return meanValue;
}
```

Arrays are passed to functions as constants
Functions can't return arrays

Arrays of Enumerated Types

```
const int HowMany = 15;
typedef enum {cent, two_cents, five_cents, ten_cents,
              twenty_cents, half_euro, euro } tCoin;
typedef tCoin tChange[HowMany];
string toString(tCoin coin); // Corresponding string
   // In main()...
   tChange pocket; // Exactly HowMany coins
   pocket[0] = euro;
   pocket[1] = five_cents;
   pocket[2] = half_euro;
   pocket[3] = euro;
   pocket[4] = cent;
   ...
   for (int coin = 0; coin < HowMany; coin++)
      cout << toString(pocket[coin]) << endl;
```


Fundamentals of Programming I

Searching in Arrays

Searching in Arrays

searcharray.cpp

Which day were sales higher than 1,000 €?

```
const int Days = 365; // Not a leap year
typedef double tSales[Days];

int search(const tSales sales) { // First element > 1000 or -1
   bool found = false;
   int ind = 0;
   while ((ind < Days) && !found) // Search Scheme
      if (sales[ind] > 1000)
         found = true;
      else
         ind++;
   if (!found) // If not found, -1 is returned
      ind = -1;
   return ind;
}
```


Fundamentals of Programming I

Capacity and Copy of Arrays

Array Capacity

The capacity of an array can't be changed during execution

Array length is a design decision:

- ✓ Sometimes it will be easy (days of a week)
- ✓ When it can be different from one execution to other, an estimated maximum length should be used
  Neither too short, nor with a lot of free space

Dynamic arrays (FP2) can be created *on the fly*

C++ STL (*Standard Template Library*):
More efficient collections with the length dynamically managed

Array Copy

An array can't be copied using the assignment operator:

```
array2 = array1; // ELEMENTS ARE NOT COPIED!!!
```

We have to copy the array element by element:

```
for (int i = 0; i < N; i++)
   array2[i] = array1[i];
```

Not Fully Used Arrays

Luis Hernández Yáñez

Not Fully Used Arrays

We may not need all the positions in an array...

The length of the array will be the maximum number of elements

There may be fewer elements than the maximum

We need an element counter...

```
const int Max = 100;
typedef double tArray[Max];
tArray list;
int counter = 0;
```

`counter`: indicates how many positions are used

We will only access positions between 0 and `counter-1`

The rest of the positions don't contain program information

Luis Hernández Yáñez

Not Fully Used Arrays

list.cpp

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

const int Max = 100;
typedef double tArray[Max];
double mean(const tArray list, int count);

int main() {
   tArray list;
   int counter = 0;
   double value, m;
   ifstream file;
   file.open("list.txt");
   if (file.is_open()) {
      file >> value;
      while ((value != -1) && (counter < Max)) {
         list[counter] = value;
         counter++;
         file >> value;
      }
```

.../...

Not Fully Used Arrays

```
      file.close();
      m = mean(list, counter);
      cout << "Mean of list elements: " << m << endl;
   }
   else
      cout << "File couldn't be opened!" << endl;

   return 0;
}

double mean(const tArray list, int count) {
   double meanValue, total = 0;
   for (int ind = 0; ind < count; ind++)
      total = total + list[ind];
   meanValue = total / count;
   return meanValue;
}
```

We will end at count-1

Promote Open Culture!

Creative Commons License

Attribution
You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Non commercial
You may not use this work for commercial purposes.

Share alike
If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

Click on the upper right image to learn more...