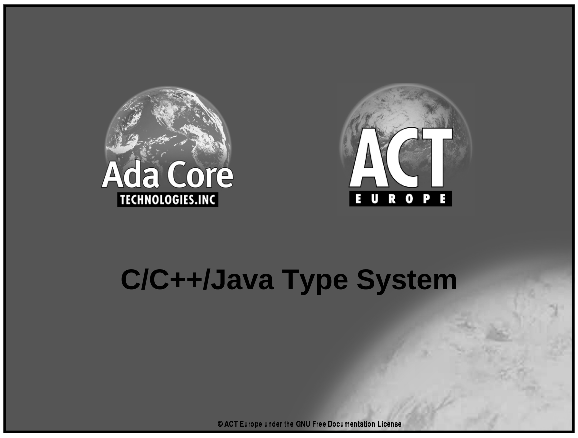
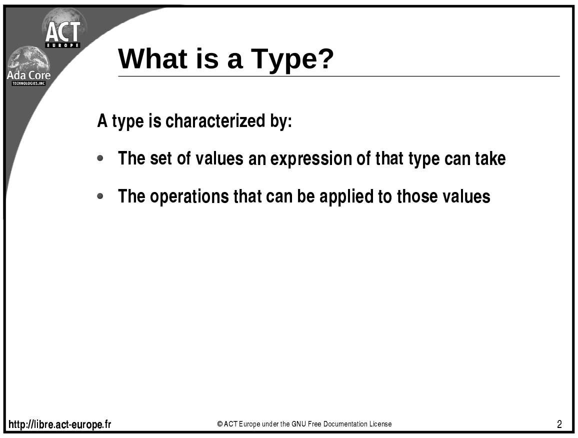


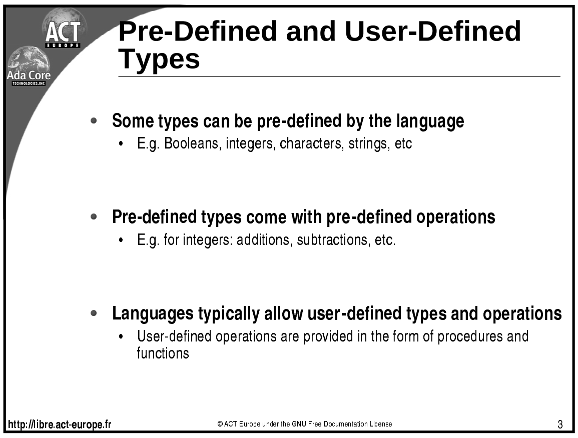
C/C++/Java Type System



What is a Type?

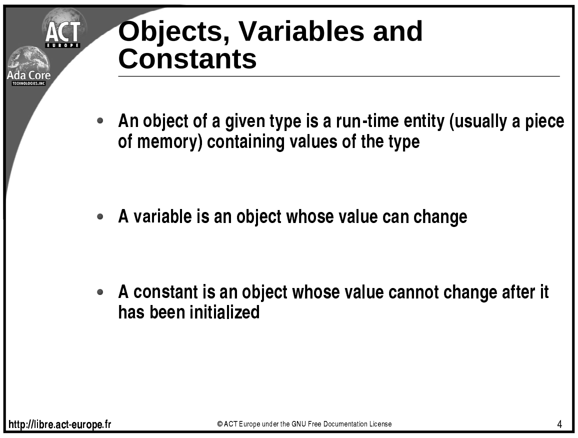
A type is characterized by:

- The set of values an expression of that type can take
- The operations that can be applied to those values



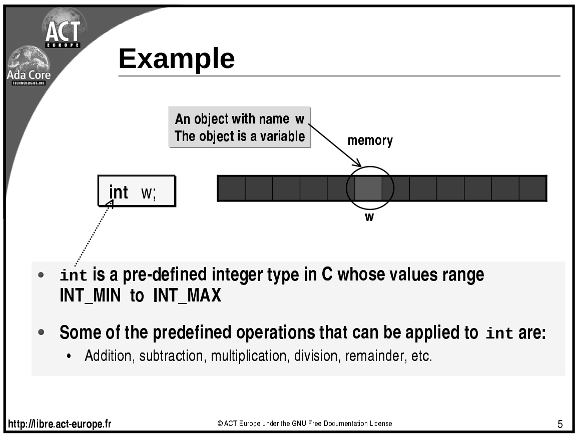
Pre-Defined and User-Defined Types

- Some types can be pre-defined by the language
  - E.g. Booleans, integers, characters, strings, etc
- Pre-defined types come with pre-defined operations
  - E.g. for integers: additions, subtractions, etc.
- Languages typically allow user-defined types and operations
  - User-defined operations are provided in the form of procedures and functions



Objects, Variables and Constants

- An object of a given type is a run-time entity (usually a piece of memory) containing values of the type
- A variable is an object whose value can change
- A constant is an object whose value cannot change after it has been initialized



Example

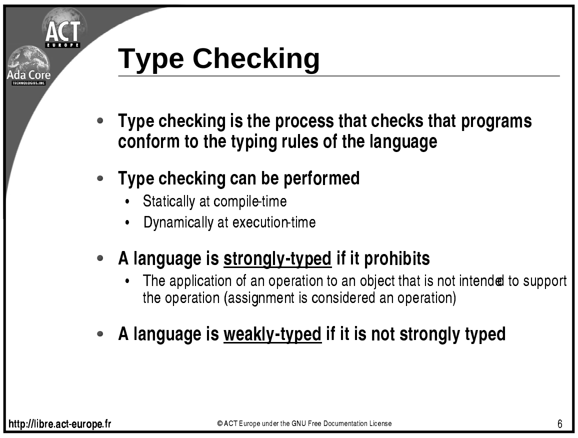
An object with name w
The object is a variable

memory

```
int w;
```

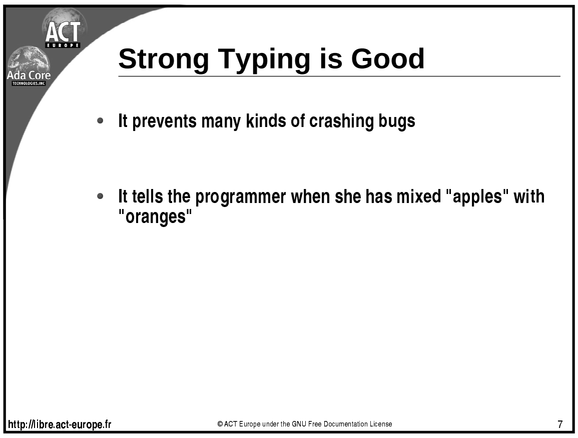
w

- `int` is a pre-defined integer type in C whose values range INT_MIN to INT_MAX
- Some of the predefined operations that can be applied to `int` are:
  - Addition, subtraction, multiplication, division, remainder, etc.



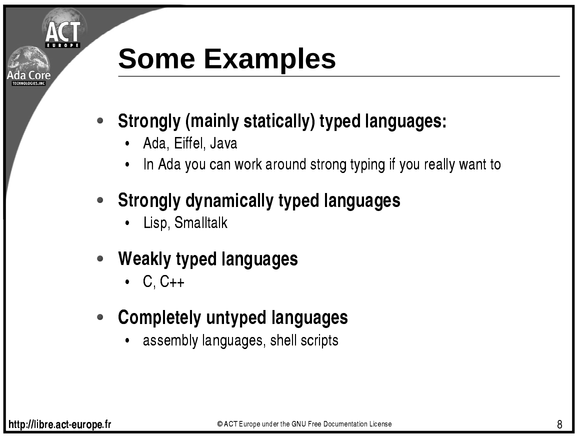
Type Checking

- Type checking is the process that checks that programs conform to the typing rules of the language
- Type checking can be performed
  - Statically at compile-time
  - Dynamically at execution-time
- A language is strongly-typed if it prohibits
  - The application of an operation to an object that is not intended to support the operation (assignment is considered an operation)
- A language is weakly-typed if it is not strongly typed



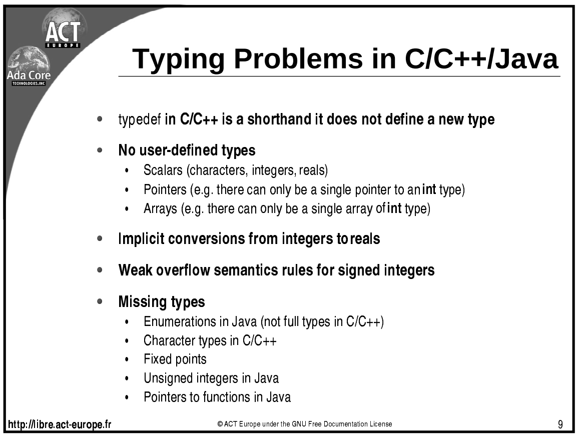
Strong Typing is Good

- It prevents many kinds of crashing bugs
- It tells the programmer when she has mixed "apples" with "oranges"



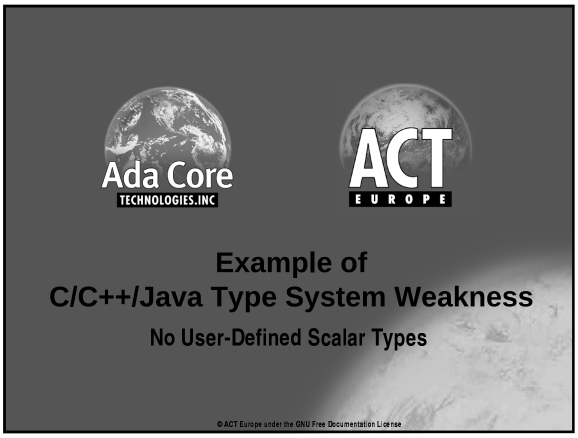
Some Examples

- Strongly (mainly statically) typed languages:
  - Ada, Eiffel, Java
  - In Ada you can work around strong typing if you really want to
- Strongly dynamically typed languages
  - Lisp, Smalltalk
- Weakly typed languages
  - C, C++
- Completely untyped languages
  - assembly languages, shell scripts



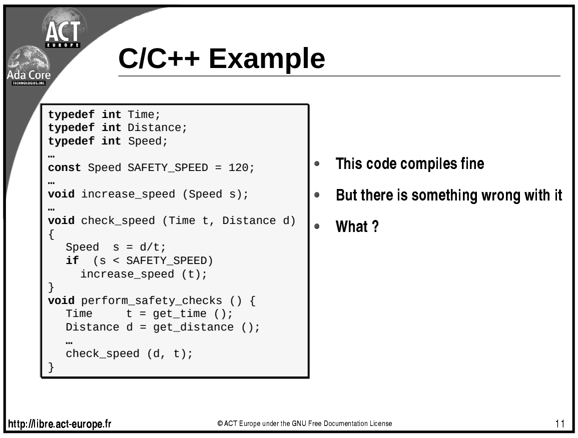
Typing Problems in C/C++/Java

- typedef **in C/C++ is a shorthand it does not define a new type**
- **No user-defined types**
  - Scalars (characters, integers, reals)
  - Pointers (e.g. there can only be a single pointer to an **int** type)
  - Arrays (e.g. there can only be a single array of **int** type)
- **Implicit conversions from integers to reals**
- **Weak overflow semantics rules for signed integers**
- **Missing types**
  - Enumerations in Java (not full types in C/C++)
  - Character types in C/C++
  - Fixed points
  - Unsigned integers in Java
  - Pointers to functions in Java



Example of C/C++/Java Type System Weakness

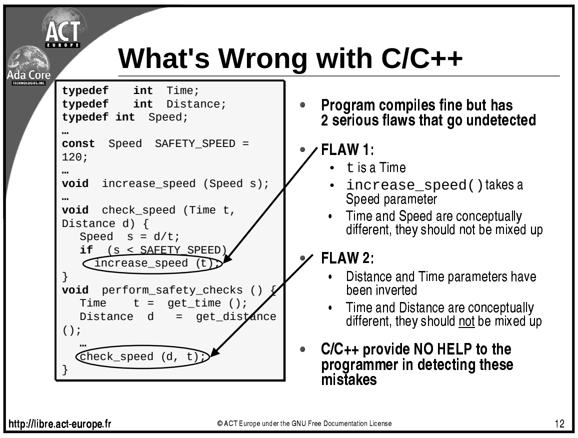
No User-Defined Scalar Types



C/C++ Example

```
typedef int Time;
typedef int Distance;
typedef int Speed;
…
const Speed SAFETY_SPEED = 120;
…
void increase_speed (Speed s);
…
void check_speed (Time t, Distance d)
{
   Speed  s = d/t;
   if  (s < SAFETY_SPEED)
     increase_speed (t);
}
void perform_safety_checks () {
   Time     t = get_time ();
   Distance d = get_distance ();
   …
   check_speed (d, t);
}
```

- This code compiles fine
- But there is something wrong with it
- What ?



What's Wrong with C/C++

```
typedef    int  Time;
typedef    int  Distance;
typedef int  Speed;
…
const  Speed  SAFETY_SPEED =
120;
…
void  increase_speed (Speed s);
…
void  check_speed (Time t,
Distance d) {
   Speed  s = d/t;
   if  (s < SAFETY_SPEED)
    increase_speed (t);
}
void  perform_safety_checks () {
   Time    t =  get_time ();
   Distance  d   =  get_distance
();
   …
  check_speed (d, t);
}
```

- **Program compiles fine but has 2 serious flaws that go undetected**
- **FLAW 1:**
  - `t` is a Time
  - `increase_speed()` takes a Speed parameter
  - Time and Speed are conceptually different, they should not be mixed up
- **FLAW 2:**
  - Distance and Time parameters have been inverted
  - Time and Distance are conceptually different, they should <u>not</u> be mixed up
- **C/C++ provide NO HELP to the programmer in detecting these mistakes**

Things are Even Worse in Java

- **There are no** typedef **in Java**
- **Everything must be an** int
- typedef **are useful for documentation purposes**
- typedef **could be used to perform sanity checks during code walkthroughs or with simple tools**
- **This problem is particularly severe in Java given that many API calls have several indistinguishable** int **parameters:**
  - AdjustmentEvent (Adjustable source, int id, int type, int value)

```
final int  SAFETY_SPEED = 120;
…
void  check_speed (int t, int d)
{
   int  s = d/t;
   if  (s < SAFETY_SPEED)
     increase_speed (t);
}
void  increase_speed (int s) { …
}
void  perform_safety_checks () {
   int   t =  get_time ();
   int  d  =  get_distance ();
   …
   check_speed (d, t);
}
```

Example of C/C++/Java Type System Weakness

Signed Integer Overflow Semantics

Overflow in C/C++/Java

```
#include <limits.h>

void  compute () {
  int  k = INT_MAX;

  k = k + 1;
}
```

- **In C/C++ signed integer overflow is undefined, anything can happ en**
  - All known implementations "wrap around"
- **In Java wrap around semantics are part of the language**

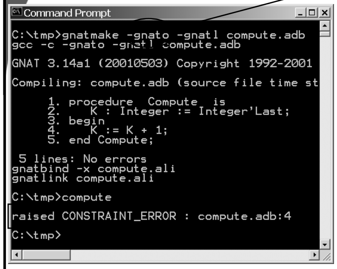
Overflow in Ada

```
procedure Compute is
   K : Integer := Integer'Last;
begin
   K := K + 1;
end Compute;
```

Exception raised at execution time

- **EVERY time there is an integer overflow in Ada an exception is r aised**

Example: Overflow in Action in Ada



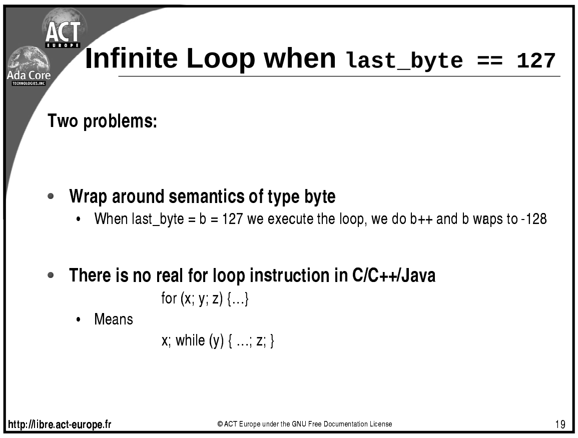
```
C:\tmp>gnatmake -gnato -gnatl compute.adb
gcc -c -gnato -gnatl compute.adb

GNAT 3.14a1 (20010503) Copyright 1992-2001

Compiling: compute.adb (source file time st

     1. procedure  Compute  is
     2.    K : Integer := Integer'Last;
     3. begin
     4.    K := K + 1;
     5. end Compute;

 5 lines: No errors
gnatbind -x compute.ali
gnatlink compute.ali

C:\tmp>compute

raised CONSTRAINT_ERROR : compute.adb:4

C:\tmp>
```

- **In GNAT you have to use the switch** -gnato **to ask for integer overflow checking**

The Badness of Wrap-Around Semantics: A Java Example

```
final  int  RADIO_PORT = …;

void open (int port) {…}
void send (int port, byte data) {…}
void close (int port) {…}

void  send_bytes (byte first_byte,
byte last_byte) {
  open (RADIO_PORT);
  for  (byte b = first_byte;
        b <= last_byte;  b++) {
    send (RADIO_PORT, b);
  }
  close (RADIO_PORT);
}
```

- **The program to the left compiles fine, and runs ...**
- **... But there is something wrong with it. What ?**



Infinite Loop when `last_byte == 127`

Two problems:

- **Wrap around semantics of type byte**
  - When last_byte = b = 127 we execute the loop, we do b++ and b waps to -128
- **There is no real for loop instruction in C/C++/Java**

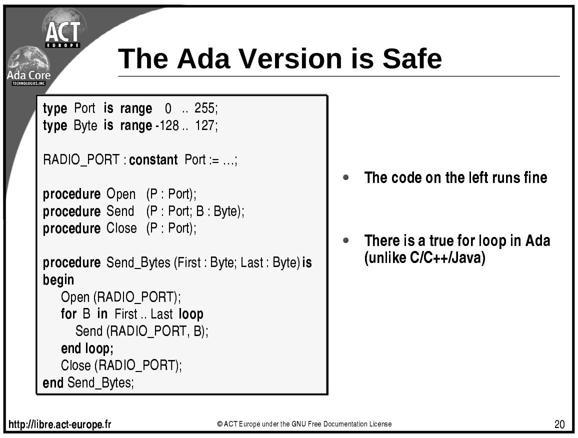
      for (x; y; z) {…}

  - Means

      x; while (y) { …; z; }

http://libre.act-europe.fr

19



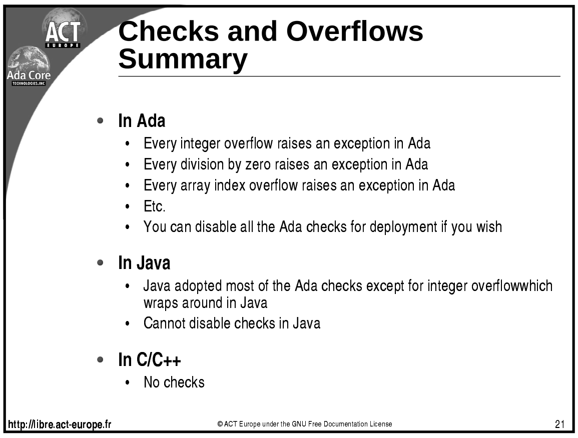
The Ada Version is Safe

```
type Port is range   0 .. 255;
type Byte is range -128 .. 127;

RADIO_PORT : constant Port := …;

procedure Open  (P : Port);
procedure Send  (P : Port; B : Byte);
procedure Close (P : Port);

procedure Send_Bytes (First : Byte; Last : Byte) is
begin
   Open (RADIO_PORT);
   for B in First .. Last loop
      Send (RADIO_PORT, B);
   end loop;
   Close (RADIO_PORT);
end Send_Bytes;
```

- **The code on the left runs fine**
- **There is a true for loop in Ada (unlike C/C++/Java)**

http://libre.act-europe.fr

20



Checks and Overflows Summary

- **In Ada**
  - Every integer overflow raises an exception in Ada
  - Every division by zero raises an exception in Ada
  - Every array index overflow raises an exception in Ada
  - Etc.
  - You can disable all the Ada checks for deployment if you wish
- **In Java**
  - Java adopted most of the Ada checks except for integer overflowwhich wraps around in Java
  - Cannot disable checks in Java
- **In C/C++**
  - No checks

http://libre.act-europe.fr

21