ECUE apprentissage de la programmation - the c++ language

Valérie Roy et Basile Marchand
Mines ParisTech

Plan

1. the core-language
   - Fundamental types in c++
   - Types
   - boolean
   - integral types
   - floating-point literals and floating-point types
   - void
   - typeid
   - Pointers
   - character literals and character types
   - const

Plan

Fundamental types in c++

Plan

Types

Every **identifier** has a **type** associated with it [a]

a. Notice that an object can have several types

the **type** determines :

- what **operations** can be **applied** to the identifier
- how the operations are **interpreted**

```
#include <iostream>
int main () {
  float f = 1;
  float g = 3;
  std::cout << f/g << std::endl;
}
$$ g++ file.cpp
$$ ./a.out
0.333333
```

```
#include <iostream>
int main () {
  int i = 1;
  int j = 3;
  std::cout << i/j << endl;
}
$$ g++ file.cpp
$$ ./a.out
0
```

Plan

1. the core-language
   - Fundamental types in c++
   - Types
   - boolean
   - integral types
   - floating-point literals and floating-point types
   - void
   - typeid
   - Pointers
   - character literals and character types
   - const

the type bool

two **values** : true and false

operations on boolean values : and (&&), or (||), **not** (!)

```
#include <iostream>
int main () {
  bool b1 = true,
       b2 = not b1,
       b3 = ! b2,
       b4 = b1 or b2,
       b5 = b1 || b2,
       b6 = b1 and b3,
       b7 = b1 && b3;
  std::cout << std::boolalpha  // boolean format
            << b1 << " " << b2 << " " << b3 << " " << b4 << " "
            << b5 << " " << b6 << " " << b7 << " " << std::endl;
}
$$ g++ file.cpp -o toto
$$ ./toto
```

the type bool

`bool` is the **type** returned by a **condition** in an `if` or an **iteration statement**

values of type `bool` can be implicitly **promoted** in integer values :by definition `true` is `1` and `false` is `0`

integer values can be implicitly converted to boolean values : nonzero integers convert to `true` and `0` converts to `false`

```
int main () {
    int i = -12;
    int j = 0;
    if (i and not j) {
        j = j + 1;
    }
    // What is the value of j ?
    std::cout << j ;
    return 0;
}
```

Plan

1. the core-language
    - Fundamental types in c++
    - Types
    - boolean
    - integral types
    - floating-point literals and floating-point types
    - void
    - typeid
    - Pointers
    - character literals and character types
    - const

Integer types

five standard `signed` **integer types** :

- `signed char`
- `short` (for `short int`)
- `int`
- `long` (for `long int`)
- `long long` (for `long long int`)

each **type provides at least as much storage** as those **preceding it** :
`signed char` $\leq$ `short int` $\leq$ `int` $\leq$ `long int` $\leq$ `long long int`

for **each** `signed integer` **type**, there **exists** an `unsigned integer` **type**

```
int main () {
  short int j {};
  long int m {};
  long long int h {};
}
```

```
int main () {
  unsigned short int l = 0;
  unsigned long int n = 0;
  unsigned long long int f = 0
}
```

each of which **occupies** the **same amount** of **storage** as the **corresponding** `signed integer` type

Integer **overflow**

you must be **careful** with **integer overflow**

```
#include<iostream>
int main () {
  using namespace std;
  unsigned int i {1};
  while (i > 0)
    i = i+1;
  cout << i-1 << " + 1 " << " = " << i << endl;
}
$$ ./a.out
4294967295 + 1  = 0
```

you must always **master** what you are **doing**

Integer literal

the type of an integral is signed by **default** (a **suffix** can specify its **type**)

unsigned int-**suffix** : u U

```
#include <iostream>
int main () {
  std::cout << 12u << 71U;
}
```

long int-**suffix** : l L

```
#include <iostream>
int main () {
  std::cout << 12l << 71L;
}
```

long long int-**suffix** : ll and LL

```
#include <iostream>
int main () {
  std::cout << 12ll << 71LL;
}
```

Maximum and minimum values of integral types : <limits>

```cpp
#include <limits>
#include <iostream>
int main () {
  using namespace std;
  cout << "int["
       << numeric_limits<int>::min() << ",␣"
       << numeric_limits<int>::max() << "]" << endl;
  cout << "unsigned␣int["
       << numeric_limits<unsigned int>::min() << ",␣"
       << numeric_limits<unsigned int>::max() << "]" << endl;
  cout << "long␣int␣["
       << numeric_limits<long int>::min() << ",␣"
       << numeric_limits<long int>::max() << "]" << endl;
  cout << "unsigned␣long␣int␣["
       << numeric_limits<unsigned long int>::min() << ",␣"
       << numeric_limits<unsigned long int>::max() << "]" << endl;
}
```

```
int           [-2147483648, 2147483647]
unsigned int  [0, 4294967295,]
long int      [-9223372036854775808, 9223372036854775807]
unsigned long int [0, 18446744073709551615]
```

representing integral types in computer

representation of integral types is bounded in computer :

- an integral type is represented using a given number of bits (n)

signed and unsigned versions of the same integral type are encoded on the same number of bytes

one bit is reserved for the sign (for example the left most bit)

- 0 for positive numbers
- 1 for negative numbers

representation is easy for positive numbers (**we take** $n = 4$ **bits**) :

- $0000 = 0$, $0001 = 1$, $0010 = 2$, ..., $0111 = 7$

representing negative integral numbers in computer

What if, for the negative numbers, we just change the sign of positive numbers ?

- $1000 = 0$, $1001 = -1$, $1010 = -2$, ..., $1111 = -7$

first we have two zeros $0000 = 0$ and $1000 = 0$ (it is not important)

more problematic $0001 + 1001 = 1010$ i.e. $1 + -1 = -2$

usual addition won't work anymore !

... but what works is $0001 + 1111 = 0000$ i.e. $1 + -1 = 0$

it is the *two*^{n}'s complement !

it was suggested in 1945 by **John von Neumann** for the **ENIAC** computer

two's complement

the two's complement of i with respect to 2^n (see wikipedia) is $2^n - i$ (i.e. $-i$) [a]

a. note that 2^n is truncated to 0 on n bits

for $i = 0010 = 2$, the two's complement is 1110 (10000 [a] $= 0010 + 1110$)

a. truncated to 000

i.e. to reverse the sign, you take the two's complement (you can represent values from -2^{n-1} to $2^{n-1} - 1$)

int	binary	2's comp.	
0	0000	0000	
1	0001	1111	10000 - 0001
2	0010	1110	10000 - 0010
...	...	...	...
6	0110	1010	10000 - 0110
7	0111	1001	10000 - 0111
-8	1000	10000	0000 - 1000
-7	1001	0111	10000 - 1001
...	...	...	...
-3	1101	0011	10000 - 1101
-2	1110	0010	10000 - 1110
-1	1111	0001	10000 - 1111

on a **signed** 4 bits integer :

- 0111 + 0001 = 1000 i.e. 7 + 1 = -8
- 1000 + 1111 = 0111 i.e. -8 + -1 = 7

What are the <limits> of integral types ?

<limits> is a library for characteristics of arithmetic types

it contains a class numeric_limits that indicates the min and max value for integral types

```
#include <limits>
int main () {
    short min = std::numeric_limits<short>::min(); // -32768
    short max = std::numeric_limits<short>::max(); // 32767
    return 0;
}
```

notice that the class numeric_limits takes a type as argument (here short) it is a template class

Plan

1. the core-language

floating point literal

a **floating point** is an **approximation** of a **real number** [a]

a. see https ://www.binaryconvert.com/

```
#include <iostream>
int main () {
  std::cout << 12. // 12
            << 12.3 // 12.3
            << 12.3e-7 // 1.23e-06
            << 12e2 // 1200
            << .7E-10; // 7e-11
   return 0;
}
```

a **floating point literal** is **composed** of :

- an **integer part**
- a **decimal point**
- a **fraction part**
- an **optional integer exponent introduced** by `e` or `E`

floating point types

three types : `float`, `double`, and `long double`

```
#include <iostream>
int main () {
  float f {12.};
  std::cout << sizeof(f);
  double d {12.};
  std::cout << sizeof(d);
  long double l {12.};
  std::cout << sizeof(l);
}
```

```
$$ ./a.out
4 (bytes)
8 (bytes)
16 (bytes)
$$
```

each **type provides at least as much storage** as those **preceding it** :
`float` $\leq$ `double` $\leq$ `long double`

floating point literal

a **suffix** can specify its **type**

`f` and `F` for `float`

```
#include <iostream>
int main () {
  std::cout << 12.f
      }
```

without suffix the default is `double`

```
#include <iostream>
int main () {
  std::cout << 12.;
}
```

`l` and `L` for `long double`

```
#include <iostream>
int main () {
  std::cout << 12.l;
}
```

What are the <limits> of floating point types ?

<limits> is a library for characteristics of arithmetic types

for floating point, numeric_limits indicates two extreme values :

- min is the smallest positive value (not 0)
- -min is the greatest negative value (not 0)
- max is the greatest positive value
- -max is the lowest negative value (or lowest)

```
          #include < limits >
#include <iostream>
int main () {
   std::cout << std::numeric_limits<float>::min();          // 1.17549e-38
   std::cout << std::numeric_limits<float>::max();          // 3.40282e+38
   std::cout << std::numeric_limits<float>::lowest ();      // -3.40282e+38

   std::cout << std::numeric_limits<double>::min();         // 2.22507e-308
   std::cout << std::numeric_limits<double>::max();         // 1.79769e+308
   std::cout << std::numeric_limits<long double>::min();    // 3.3621e-4932
   std::cout << std::numeric_limits<long double>::max();    // 1.18973e+4932
   return 0;
}
```

Maximum and minimum values of floating point types <limits>

```cpp
#include <limits>
#include <iostream>

int main () {
  using namespace std;
  cout << "float["
    << (float)(numeric_limits<float>::min()) << ", "
    << (float)(numeric_limits<float>::max()) << "]" << endl;
  cout << "double["
    << (double)(numeric_limits<double>::min()) << ", "
    << (double)(numeric_limits<double>::max()) << "]" << endl;
  cout << "long double["
    << (long double)(numeric_limits<long double>::min()) << ", "
    << (long double)(numeric_limits<long double>::max()) << "]" << endl;
}
```

```
$$ g++ -std=c++11 file.cpp
$$ ./a.out
float       [1.17549e-38,  3.40282e+38]
double      [2.22507e-308, 1.79769e+308]
long double [3.3621e-4932, 1.18973e+4932]
$$
```

Plan

1. the core-language
   - Fundamental types in c++
   - Types
   - boolean
   - integral types
   - floating-point literals and floating-point types
   - void
   - typeid
   - Pointers
   - character literals and character types
   - const

void

The `void` type is an **incomplete type** that has an **empty set** of **values**

used as **return type** for **functions** that **do not return** a value

(optional) used as a **parameter** for **functions** that do **not take** any **argument**

```
void foo(void) {}

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

it is the **base type** for **pointers** to **objects** of **unknown type**

```
int main () {
  void *p1 = 0;
  void *p2 = nullptr; // c++11
}
```

```
int main () {
  void *p1 = 0,
       *p2 = nullptr; // c++11
}
```

Plan

1. the core-language
   - Fundamental types in c++
   - Types
   - boolean
   - integral types
   - floating-point literals and floating-point types
   - void
   - typeid
   - Pointers
   - character literals and character types
   - const

typeid

the `typeid` operator returns, at execution time, information on the type of an object

it can be used for static type identification

you must include the header `<typeinfo>` [a]

a. otherwise your program is *ill-formed*

it returns a `std::type_info` object representing the type of your object

```
#include <iostream>
#include <typeinfo>

int main () {
  std::cout << typeid(true).name() // b
            << typeid('c').name()  // c
            << typeid(12).name()   // i
            << typeid(int*).name() // Pi
            << typeid(short*).name() // Ps
            << typeid(17.5).name()  // d
            << typeid(17.5f).name();// f
    return 0;
}
```

Plan

1. the core-language

What is a pointer ?

a pointer is the **address** of an object in memory

for a **type** `T`, `T*` is the type **pointer to** `T`

in an expression :

- **&** is the **address of** extttoperator (it returns the address of an object)
- ***** is the **object pointed by** extttoperator (it returns the object at the given address)

the **implementation** of **pointers** is directly **bound** to the **addressing mechanisms** of the **machine**

`nullptr` is the null pointer

```
int main () {
  int i = 78;   // an integer

  int* pi = &i;  // the address of the integer i

  (*pi) = 12;    // the object pointed by pi
                 // is assigned with 12

   return 0;
}
```

nullptr

the literal `nullptr` represents the **null** pointer [a]

a. i.e. a pointer that **does not** point to an object

`nullptr` can be assigned to any pointer type [a]

a. but only to pointer

there is **only** one `nullptr` shared by all pointer types [a]

a. not a null pointer for each pointer type

before `nullptr`, zero 0 was used as a notation for the null pointer

```
int main () {
  float* pf {};          // nullptr by default
  char* pc = nullptr;    // nullptr is explicit
  int* pi = 0;           // ok (will be nullptr)
  bool* pb;              // not ok ! pb is uninitialized
}
```

Plan

1. the core-language
   - Fundamental types in c++
   - Types
   - boolean
   - integral types
   - floating-point literals and floating-point types
   - void
   - typeid
   - Pointers
   - character literals and character types
   - const

ASCII character literals in c++

a character literal in c++ is **one** character enclosed in **single quotes**

an ordinary character literal has type `char`

```
#include <iostream>
int main () {
  std::cout << 'a' << '\t' << 'b' << '\n';
  return 0;
}
```

non ascii character literals in c++

characters may be **optionally preceded** by `u`, `U`, a character literal that **begins** with the letter :

- `u` has **type** `char16_t` [a]
- `U` has **type** `char32_t`
- `L` (wide-character) has **type** `wchar_t`

a. the two new character types : `char16_t` and `char32_t` are designed to **deal with** non-ascii **character encoding**

```
#include <iostream>
int main () {
  std::cout << u'a' << U'b' << L'c';
  return 0;
}
```

What does the c++ standard say about character types

the type `char` shall be **large** enough to store ansii characters

`char`s are almost universally considered **8-bit long type**, a `char` *can hold* $2^8 = 256$ values

with n **bits** :

unsigned type range	$[0 \text{ to } 2^n - 1]$	from 0 to 255
signed type range	$[-2^{n-1}, 2^{n-1} - 1]$	from -128 to $+127$

it is **implementation-defined** whether a `char` is signed or not

thus it is **not safe** to **assume** that `char` can hold **more than 127 characters**

```
int main () {
  char c = 128;   // is this code portable ?
  return 0; }
```

What does the c++ standard say about character types

the type `char` shall be **large** enough to store ansii characters

`char`s are almost universally considered **8-bit long type**, a `char` *can hold* $2^8 = 256$ values

with n **bits** :

unsigned type range	$[0 \text{ to } 2^n - 1]$	from 0 to 255
signed type range	$[-2^{n-1}, 2^{n-1} - 1]$	from -128 to $+127$

it is **implementation-defined** whether a `char` is signed or not

thus it is **not safe** to **assume** that `char` can hold **more than 127 characters**

```
int main () {
  char c = 128;   // is this code portable ?
  return 0; }
```

no it supposes that char are unsigned otherwise it overflows, plain char values outside $[0, 127]$ lead to portability problems

What does the c++ standard say about character types

characters can be **explicitly** declared `unsigned` or `signed`

```
int main () {
    unsigned char c1 = 128;
    signed char c2 = -128;
    return 0;
}
```

plain `char`, `signed char`, and `unsigned char` are three **distinct** types

but they occupy the **same amount** of storage

Maximum and minimum values of character types <limits>

```
#include <iostream>
#include <limits>
int main () {
  std::cout << (int) std::numeric_limits<char>::min();    // -128
  std::cout << (int) std::numeric_limits<char>::max();    // 127
  std::cout << (int) std::numeric_limits<unsigned char>::max();    // 255
  std::cout << (int) std::numeric_limits<char16_t>::max();         // 65535
  std::cout << (long int) std::numeric_limits<char32_t>::max();    // 4294967295
}
```

example of ascii characters and its ascii representation

- we take the character 'G'
- we print its ascii representation

```
#include <iostream>
int main () {

  char G = 'G';         // the character G
  std::cout << "the␣character␣" << G
            << std::endl;

  int Gint = G;         // its ascii representation 71
  std::cout << "its␣ascii␣representation␣(in␣decimal)␣" << Gint <<
    std::endl;

}
```

```
$ g++ ascii_G.cpp -o out
$ ./out
the character G
its ascii representation (in decimal) 71
```

example of ascii character and its hexadecimal, octal and binary representations

we write the character 'G' in decimal, in hexa-decimal, in octal and in binary

```
#include <iostream>
#include<bitset>
int main () {
  char G = 'G';        // the character G
  int Gdec = 71;       // decimal ascii code of 'G' (decimal is 0 to 9)
  int Ghexa = 0x47;    // hexa-decimal ascii code (hexa is 0 to F)
  int Goctal = 0107;   // octal ascii code (octal 0 to 7)
  int Gbin = 0b1000111; // binary ascii code (bin is 0 to 1)
  // (std::hex modifies the current numeric base, here decimal, for integer output)
  std::cout << G     << std::endl;             // G
  std::cout << Gdec  << std::endl;             // 71
  std::cout << std::hex << Ghexa << std::endl;  // 47
  std::cout << std::oct << Goctal << std::endl; // 107
  // to print binary number (for example)
  std::cout << std::bitset<8>(G)  << std::endl; // 01000111
  return 0;
}
```

Exercice Capitalization of ASCII Letters

```
#include<iostream>
// implement the upper function
int main () {
   char c = 'a';
   std :: cout << upper(c) << std :: endl;
   return 0;
}
```

- the upper function capitalizes the ascii letter passed as argument
- in the ascii encoding, the letters are contiguous (from 'a' to 'z' and from 'A' to 'Z')
- in the ascii encoding, the distance between 'a' and 'A' is 'a' - 'A'

Exercice Correction : Capitalization of ASCII Letters

- 'a' - 'A' is the distance between uppercase letters and lowercase letters
- by removing this distance to a lowercase letter, we obtain its uppercase counterpart

```
// when c is a lowercase letter , the
// function returns the letter capitalized
// otherwise it returns c
char upper (char c) {
   if (c >= 'a' and c <= 'z' )
      return c - ('a' - 'A' );
   return c;
}
```

```
#include<iostream>
int main () {
   char c;
   do {
      std :: cin >> c;
      std :: cout << upper(c)
                  << std :: endl;
   } while (c != '0' ); // type '0' to stop
   return 0;
}
```

Plan

1. the core-language
   - Fundamental types in c++
   - Types
   - boolean
   - integral types
   - floating-point literals and floating-point types
   - void
   - typeid
   - Pointers
   - character literals and character types
   - const

constant objects of built-in types

when the value of an object does **not** need to be **modified** after initialization : declare the object as being `const`

when in a function, a **parameter** is **read** but never **written** : pass it `const` to the function

`const` for read-only function's arguments is a very **good** programing style

it **prevents** programmers to introduce bugs