ECUE apprentissage de la programmation - the c++ language

Valérie Roy et Basile Marchand
Mines ParisTech

Plan

1. the core-language
    - Arrays

Plan

1. the core-language
    - Arrays

Arrays

Arrays in memory

in memory, an array is a contiguous **sequence** of objects of the **same type**

an array is represented by the **address** of its **first element** (if any)

```
int main () {
  int tab [3];
  tab[0] = 42;
  tab[1] = 17;
  tab[2] = 81;
  return 0; }
```

the type of `tab` is **pointer** to an `integer`

`tab[i]` is equivalent to `*(tab+i)` :

- we consider the address `tab`
- *i* is the number of element to pass
- i.e. we add to the address of the array, $i\times$ the size of an element of the array (here `int`)
- `(tab+i)` is the address of the $i+1$ element

```
tab              tab+1            tab+2
|                |                |
v        0       v       1        v       2
|----------------|----------------|----------------|
|       42       |       17       |       81       |
|----------------|----------------|----------------|
  <-sizeof(int)->  <-sizeof(int -> <-sizeof(int)->
```

Arrays in memory

`T tab [ n ]`

in an **expression** `tab` is **converted** to a **pointer** to the **first element** of the **array**

in an **expression** `*tab` is the **first element** of the **array**

in an **expression** `tab+i` is a **pointer** to the $i+1$ **element** of the **array** (if any !)

```
int main () {
  int tab [3] { 42, 17, 81 };
  sizeof(tab)/sizeof(int);  // number of ints
  tab[0];      // the first  element
  *tab;        // the first  element
  tab[1];      // the second element
  *(tab+1);    // the second element
  int i = 2;
  tab[i];      // the i+1    element
  *(tab+i);    // the i+1    element
}
```

Use of arrays

an array is the **solution** for **simple** fixed-length sequences of objects of a given type

array is a **low-level** facility[a]

[a] to be used inside higher-level implementations such as `std::string`, `std::vector`, ...

prefer using the c++ standard library facilities (`std::string`, `std::vector`, ...)

Array initialization

an array can be initialized by a list of values

```
int main () {
   int tab [3] = {42, 17, 81};
   return 0;
}
```

you cannot initialize with more elements than the given size !

```
int main () {
   int tab [2] = {42, 17, 81};
   return 0;
}
```

but if you supply fewer elements than the given size, the others are 0-initialized

```
int main () {
   int tab [5] = {42, 17, 81};
   return 0;
}
```

Global arrays initialization

```
bool tab1 [12];
const float tab2 [] = { 2.4, 4e12, 0.3, -5 };

const int N = 13;
int tab3 [N];

int main () {
  tab1[11] = true;
  tab2[0] = .12; // assignment of read-only location 'tab2[0]'
  return 0;
}
```

tab1 : global array of 12 bool initialized to false [a]

tab2 : global array of 4 initialized **constant** floats

tab3 : array of N (here 13) zero-initialized integers

assignment of an element of tab2 causes a **compile-time error** because constant **cannot** be assigned they can only be initialized

[a] global variables are, by default, zero-initialized

Global array's size

the size of **global** arrays **must** be a **compile-time constant**

```
const int N = 3;
int tab[N]; // ok N is a constant integer

int M = 12;
float tabf[M]; // compile-time ERROR: M is NOT a constant !

int main () {
}
```

Local arrays

a local array is allocated in the **stack**

we do **not** need to know the size of a local array at compile-time

```
void foo () {
  bool tab1 [12];
  tab1[0] = true;
}
int main () {
  const float tab2 [] = { 2.4, 4e12, .3, -5. };
  tab2[0] = 12.; //  assignment of read-only location ’tab2[0]’
}
```

What are the values of the uninitialized elements of a local array ?

Local arrays

a local array is allocated in the **stack**

we do **not** need to know the size of a local array at compile-time

```
void foo () {
  bool tab1 [12];
  tab1[0] = true;
}
int main () {
  const float tab2 [] = { 2.4, 4e12, .3, -5. };
  tab2[0] = 12.; //  assignment of read-only location ’tab2[0]’
}
```

What are the values of the uninitialized elements of a local array ? we do not know !

sizeof of an array

the size in **bytes** of an array tab is sizeof(tab)

the size in **bytes** of an element of type T is sizeof(T)

you can know the number of elements in an array tab of element of type T : sizeof(tab) / sizeof(T)

```
#include <iostream>
int main () {
  int tab [3];

  // size in bytes of the array tab
  std::cout << sizeof(tab);  // 12

  // size in byte of an int on my computer
  std::cout << sizeof(int);  // 4

  // the number of elements in the array tab
  std::cout << sizeof(tab)/sizeof(int);  // 3
  return 0;
}
```

Problem with local arrays

```
int main () {
  int t[-50]; // COMPILE-TIME ERROR
              // (size of array 't' is negative)
}
```

```
int main () {
  int n = -50;
  int tab [n];  // RUNTIME ERROR
                // (invalid memory manipulation)
}
```

the execution of your program can be aborted with a **segmentation fault (core dumped)**

... or the execution of your program can continue a little while in a corrupted memory

ADVANCED (constexpr)

ADVANCED constexpr

remember that global array must have a constant size (a compile-time constant)

```
const int N = 13;
int t3 [N];
// OK the size is a COMPILE-TIME CONSTANT
```

i would like to use a function !

```
const int size () {
  return 20;
}
const int t4 [size()];  // ERROR the size is not a
                        // COMPILE-TIME constant
```

constexpr

in **previous c++ versions**, a **constant expression** was **not allowed** to **contain** a **function call**

you have a way to **guarantee** that an **initialization** is done at **compile time**

objects declared `constexpr` have their initializer evaluated at compile time

constexpr for array initialization

because c++ **requires** the **use** of **constant expressions** when **defining** a global **array**

C++11 introduced the **keyword** constexpr

constexpr **allows** the **user** to **guarantee** that a **function** is a **compile-time** constant

```
constexpr int size () { return 20;}
int tab[size()];
```

the **compiler understands** that size() is a **compile-time constant**

constexpr can be **used** for **non integral types**