Some pointers on pointers

Salvatore Aiola

Yale University

Analysis Tutorial
ALICE Week
CERN, July 28th, 2017

Outline

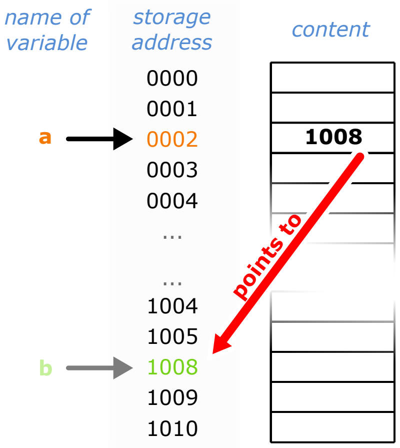
1. Introduction
   - Disclaimer
   - Pointers and Memory Management
2. Why a raw pointer is hard to love
3. Smart Pointers
4. Conclusions

Introduction

Disclaimer

- Some parts will be too boring for some people, some other parts will be too advanced for others
  - On the other hand, most people might find some parts interesting
- Not a comprehensive lecture on memory management in C++
  - Not even close!
  - Rather some suggestions to write efficient, maintainable and robust code
- Not an IT expert!
  - Experience from writing analysis code in ALICE + some limited readings

What is a Pointer?



A pointer is an object whose value “points to” another value stored somewhere else in memory

- Very powerful tool
- Great power = great responsibility!
- Extensive use of pointers in ROOT/AliRoot/AliPhysics

Using a Pointer

```cpp
/* Defining a pointer */
int* a; // declares a pointer that can point to an integer value
//DANGER: the pointer points to a random memory portion!

int* b = nullptr; // OK, pointer is initialized to a null memory address

int* c = new int; // allocate memory for an integer value in the heap
//and assign its memory address to this pointer

int** d = &a; // this pointer points to a pointer to an integer value

MyObject* e = new MyObject(); // allocate memory for MyObject
// and assign its memory address to this pointer

/* Using a pointer */
int f = *c; // dereferencing a pointer and assigning the pointed
// value to another integer variable

e->DoSomething(); // dereferencing a pointer and calling
// the method DoSomething() of the instance of MyObject
// pointed by e
```

Why a raw pointer is hard to love

Array or single value?

- A pointer can point to a single value or to an array, however its declaration does not indicate it
- Different syntax to destroy (= deallocate, free) the pointed object for arrays and single objects

```
void UserExec()
{
  MyTrack *track = new MyTrack(0,0,0,0);
  double *trackPts = new double[100];
  double *returnValue = AnalyzeTracks(trackPts);

  // here use the pointers

  delete track;
  delete[] trackPts;
  delete returnValue; // or should I use delete[] ??
}
```

Memory leaks and double deletes

- Each memory allocation should match a corresponding deallocation
- Difficult to keep track of all memory allocations in a large project
- Ownership of the pointed memory is ambiguous: multiple deletes of the same object may occur

```
void UserExec()
{
  AliVTracks* tracks = FilterTracks();

  AnalyzeTracks(tracks);

  delete[] tracks; // should I actually delete it??
  //or was it already deleted by AnalyzeTracks?
}
```

Smart Pointers

Conclusions

Final remarks

- When the extra-flexibility of a pointer is not needed, do not use it
- Alternative example: arguments by reference (not covered here)
- Avoid raw pointers whenever possible!
- Smart pointers (unique_ptr and shared_ptr) should cover most use cases and provide a much more robust and safe memory management