High Level Programming

Dynamic Memory

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

Introduction

- Our C++ programs so far have use only
  - Static memory, to store
    - Static objects
    - Static data members defined inside classes
    - Objects defined ouside any functions
  - Stack memory, to store
    - Non-static objects, such as the ones defined inside functions
- Objects allocated in the static and stack memory
  - Have a duration depending on their scope
  - Are automatically created and destroyed by the compiler

Introduction

- Programs can also use heap memory
  - The heap is used to store dynamic objects
    - The strategy directly derives from the C language
  - Dynamic objects
    - Are explicitly managed through dynamic allocation
    - Have a lifetime controlled by the program
      - Are allocated into the heap when they are created during the execution
      - Must be destroyed as soon as they are no longer necessary

Introduction

- In C++ there are several functions to manipulate memory dynamically
  - Old C-like functions
    - Malloc, calloc, and free
    - Realloc does not exist
      - It can be implemented "manually", through a malloc followed by an explicit data transfer
  - C++ function
    - **New**, to create an object
    - **Delete**, to free the memory associated with an object

Because of the constructor and destructor used in C++

Similar to malloc and calloc

Similar to free

Like in C, they manage memory **directly**

Introduction

- Main differences between C and C++ primitives

Malloc & Free	New & Delete
Return a void *.	Return a fully typed pointer.
There is no constructor (destructor) they just return raw memory.	Call the constructor (destructor) which operate on the memory before returning.
Are library functions.	Are operators.
There is no overloading.	Can be overloaded.
On failure, there are no exceptions.	On failure, they allow exceptions.
Arrays are a sequence of N-cells.	Explicitly manipulate arrays of a specific type.

New

- The **new** operator
  - Allocates a memory block compatible with the type and the size defined
    - To create the object, it calls the **constructor** of that object
  - Allocates unnamed objects
    - Thus, **new** returns a **pointer** to that object

Examples

```
using namespace std;
...

int * v1 = new int;

int * v2 = new int();

int * v3 = new int(12);

int * vect1 = new int[10];


string *ps1 = new string;
string *ps1 = new string();


auto p1 = new auto(obj);
```

Single variable with no initialization (malloc)
*v1 is undefined

Single variable initialized to 0 (calloc)
*v2 is 0

Single variable initialized to 12 (similar to calloc)
*v3 is 12

Array of size 10
vect1 cannot be nullptr

Empty string (default initializer)
String initialized to empty

p1 points to an object of type of obj, initialized from obj

New

- The are two versions of new
  - The normal version
    - When the requested memory is not available, it fires an **exception**
    - It **never** returns a **nullptr** pointer
  - The nothrow version
    - When the requested memory is not available, it returns a **nullptr**
    - It **never fires** any **exception**

Examples

```
int *p1 = new int;

int *p2 = new (nothrow) int;

int *p3 = new (nothrow) int[20];
```

If the allocation fails, new throws **std::bad_alloc**

If the allocation fails, new returns **nullptr**

Array of size 20 **or** a nullptr

- It is possible to allocate constant objects
  - The returned pointer is a pointer to const

```
const int *p4 = new cost int(100);

const my_class *p5 = new const my_class;
```

Constant objects must be initialized explicitly

Or implicitly, with the default constructor

Examples

- Like in C, a pointer does not know the number of elements to which is pointing to
  - It just knows the address of the first element

```
double * p1 = new double;

*p1 = 7.3;
p1[0] = 8.2;

p1[7] = 9.4;    // Error
p1[-4] = 2.4;   // Error
```

*p1 = 7.3;: This assigns the value 7.3 to the memory location pointed to by p1.
p1[0] = 8.2;: This is equivalent to *(p1 + 0), which also assigns 8.2 to the memory location pointed to by p1.
p1[7] = 9.4;: This attempts to access the memory location pointed to by p1 with an offset of 7 elements. However, p1 was allocated memory for only a single double value, so accessing beyond this allocated memory (out-of-bounds access) results in undefined behavior.
p1[-4] = 2.4;: Similarly, this attempts to access memory before the allocated memory block, resulting in undefined behavior.

```
double * p2 = new double[10];

*p2 = 7.3;
p2[0] = 8.2;
p2[7] = 9.4

p2[-4] = 2.4;   // Error
```

Examples

- Like in C, a pointer does not know the type of the object to which is pointing to
  - It just knows the address of the first element

```
int * p1 = new int(10);
int *p2 = p1;

float *p3 = p1;          // Error
char *p4 = p1;           // Error
```

int *p1 = new int(10);: This dynamically allocates memory for an integer on the heap and initializes it with the value 10. The pointer p1 is then assigned the memory address of this dynamically allocated integer.
int *p2 = p1;: This assigns the value of p1 (the memory address of the dynamically allocated integer) to the pointer p2. Both p1 and p2 now point to the same memory location.
float *p3 = p1;: This attempts to assign the memory address stored in p1 to the pointer p3, which is of type float *. This is a type mismatch because p1 is pointing to an integer, not a float. However, in C++, this assignment is allowed due to implicit type conversion, but it's not recommended because it can lead to erros behavior when dereferencing p3.
char *p4 = p1;: Similarly, this attempts to assign the memory address stored in p1 to the pointer p4, which is of type char *. This is also a type mismatch because p1 is pointing to an integer, not a character.
In C++, while implicit type conversions are allowed for pointers, it's important to use proper type safety and avoid such conversions unless necessary, as they can lead to unintended behavior and potential bugs in the code. It's always best to explicitly cast pointers to the correct type to ensure type safety and avoid runtime errors.

Delete

- To avoid memory exhaustion, we must return the allocated memory to the system
  - A dynamic object exists until it is explicitly deleted
- The **delete** operator free previously allocated memory
  - It calls the destructor of the object
    - For each call to **new** there should be a call to **delete**
    - For each object, we call new **first**, delete **after**
  - The are two versions of **delete**
    - **Single** for single variables and objects
    - **Multiple** for multiple blocks, such as vectors

Examples

```
int * v1 = new int;
int * v2 = new int(12);
int * vect1 = new int[10];
int * vect2 = new (nothrow) int[20];
my_class *p = new my_class;
```

Allocation with news

Subsequent deletes

```
using namespace std;

...

delete v1;
delete v2;

delete[] vect1;
delete[] vect2;

delete p;
```

Single objects

Arrays

Calls the default destructor before release

Examples

- Again, notice that
  - Dynamic objects managed through built-in pointers exist until they are explicitly deleted

```
my_type *my_func (... arg) {
  return (new (my_type(arg));
}

...
my_type *p = my_func (arg);
```

Returns a pointer to a dynamically allocated object

The caller uses *p and it is responsible to delete p

The memory referenced by p **is not automatically freed** if p gets out of scope

Dangling pointers & memory leaks

- Managing memory through new and delete is error prone as we can have
  - Dangling pointers
  - Memory leaks

Dangling pointers

- When we delete a pointer, the pointer becomes invalid
  - Although the pointer is invalid, it still points to a correct memory address
  - A pointer that do not point to a valid object of the appropriate type is call **dangling pointer**
  - Dangling pointers
    - Are generated when memory is released
    - May generate memory violations
    - Are very difficult to discover and trace-back

Memory leaks

- A dynamic object managed through a built-in pointer exists until it is explicitly deleted
  - If we forgot to delete an object, we have a memory leak
  - Many programs cannot afford memory leaks
    - If a function leaks 1 byte but it is called 10^7 times, then the program leaks 10 MBytes

Dangling pointers & memory leaks

- Standard strategies to avoid dangling pointers and memory leaks provide only limited protection
  - To avoid dangling pointers, we can set the pointer to **nullptr** when we
    - Define a pointer

```
int *ptr;
```

→

```
int *ptr = nullptr;
```

    - Delete a pointer

```
delete ptr;
```

→

```
delete ptr;
ptr = nullptr;
```

Dangling pointers & memory leaks

- Unfortunately, it is easy to introduce bugs

```
int *p = new int();
auto q = p;
...
delete p;
p = nullptr;
```

Resetting p has no effect on q that is still dangling

- Moreover, memory leaks are still present

```
void foo(unsigned length) {
  int* buffer = new int[length];
  ...
  if (condition)
    return;
  ...
  delete[] buffer;
  return;
}
```

It is hard to ensure that we free memory at the right time

RAII & smart pointers

- To systematically avoid dangling pointers and memory leaks C++ offer a few alternatives
  - Use the sequential and associative containers
    - The standard library **does not** require the explicit use of new and delete
  - Use an automatic garbage collector
    - A garbage collector keeps track of all allocations and returns the memory when it is no longer used
    - C++ implements this technique through **smart pointers**
      - A smart pointer is like a regular pointer but it **automatically deletes** the object to which it points when it gets out of scope

RAII & smart pointers

- This technique is often referred to as RAII
  - RAII = Resourse Acquisition is Initialization
  - It binds the lifetime of a **resource** to the lifetime of the corresponding **object** (pointer)
    - A resource is available during the entire lifetime of the object (pointer)
    - The resource is released when the lifetime of the object (pointer) ends
  - Encapsulates each resource into a class whose unique responsibility is to manage the resource
    - The constructor acquires each resource
    - The destructor releases each resource
  - Object should have **automatic** storage duration

RAII & smart pointers

- The C++ library introduces three diffrerent types of smart pointers
  - Shared pointer, **shared_ptr**
    - Which allows multiple pointers to refer to the same object
  - Unique pointer, **unique_pointer**
    - Which owns the pointer to which it points
  - Weak pointer, **weak_ptr**
    - A weak reference to (form of) a shared pointer (generated by a shared pointer)
  - Many operations are similar for all pointers; others are specific for each of them

Shared pointers

- Shared pointers are used when a resource may have several owners
  - Multiple pointers may refer to the same resource
    - Each dynamic resource has a **reference count** counting the number of pointers referencing it
      - The count is incremented when we copy the shared pointer
      - The count is decremented when we assign another value to the shared pointer (i.e., the shared pointer gets out of scope)
    - When the counter goes to zero, the resource is released automatically

Shared pointers

- The safest way to allocate memory is to call make_shared
  - This function allocates and initializes a resource into the heap and returns a shared pointer pointing to it
- A shared pointer may be copied and moved
  - Operations on share pointers are rather expensive
    - The corresponding counter has to be updated
  - These operations should be avoided when possible

Main operations

Smart pointers are **templates**. We must specify the type to which they point

Type	Main characteristics
shared_ptr<T> p; unique_ptr<T> p;	Define a smart pointer p that point to an object of type T. Initialize p to nullptr.
make_shared<T>(args)	Returns a shared pointer pointing to a dynamically allocated object of type T. args is used to initialize the object.
shared_ptr<T> p(q);	Pointer p is a copy of pointer q (whose counter is incremented).
p	True if p points to an object.
*p p->…	The object pointed by p.
p.use_count()	Returns the number of objects shared with p.
p.unique()	Retuns true if p.use_count() is one.
p.reset()	If p is the only shared pointer pointing to an object, frees the object.

Examples

Share pointers:
Definition

```
#include <memory>

using namespace std;

shared_ptr<string> p1;
shared_ptr<list<int>> p2;
```

!

p1 points to a string.
If we do not initialize a smart pointer, it is initialized to **nullptr**

p2 points to a list of stringsc

Share pointers:
Allocation

```
shared_ptr<int> p3 = make_shared<int>(42);


shared_ptr<string> p4 = make_shared<string>(3,9);
```

p3 points to an int equal to 42

p4 points to the string "999"

Examples

Share pointers:
Allocation with new

- The operator new can allocate shared pointers
  - The type conversion must be made explicit
  - The process is less efficient and requires two allocations
    - The first one, to allocate the required memory
    - The second one, to reserve the space for the counter

```
shared_ptr<int> p1 = new int(42);

shared_ptr<int> p2(new int(42));

shared_ptr<int> p3 = shared_pointer<int> (new int(9));
```

Implicit conversion:
Error p1 is wrong

Direct initialization:
p2 is OK

Explicit conversion:
p3 is OK

Examples

Share pointers:
Garbage collector

- Shared pointers automatically free dynamic objects when they are no longer needed

```
shared_ptr<int> p = make_shared<int>(20);
auto q = make_shared<int>(10);

p = q;
```

p now points to q.
The counter of q has been incremented.
The counter of p has been decremented.
As the object p pointed to, has no users, that object is automatically freed.

```
shared_ptr<foo> myf1 (T arg) {
  ...
  return make_shared<foo>(arg);
}
void myf2 (T arg) {
  shared_ptr<foo> p = myf1 (arg)
  ...
  use p
  ...
  return;
}
```

We return a shared pointer. Thus, we do not have to worry about deallocation

Pointer p goes out of scope. The memory to which p points is automatically freed

Examples

Share pointers:
Garbage collector

- Shared pointers automatically free dynamic objects when they are no longer needed

```
shared_ptr<foo> myf1 (T arg) {
  ...
  return make_shared<foo>(arg);
}
shared_ptr myf2 (T arg) {
  shared_ptr<foo> p = myf1 (arg)
  ...
  use p
  ...
  return (p);
}
```

Add one to the reference count to that object

Pointer p goes out of scope.
The object has a counter equal to one.
The memory is not freed

Introduced with C++11

Unique pointers

- Unique pointers represents ownership
  - A unique pointer ows the object to which it points
    - Only one unique pointer at a time can point to a given object
  - The object is automatically disposed when the unique pointer goes out of scope
  - Can be moved, not copied
    - Useful to obtain a **movable handle** for an **immovable object**
    - When used as a function parameter or a return type indicates a transfer of ownership
    - They should almost always be passed by value

Examples

Share pointers: Definition & Allocation

```
#include <memory>

void my_func() {
   std::unique_ptr<int> valuePtr(new int(15));
   ...
   if (...)
      return;
   ...
}
```

Definition and allocation

No memory leak !

```
#include <memory>

unique_ptr<int> clone1 (int p) {
  return unique_ptr<int>(new int(p));
}

unique_ptr<int> clone2 (int p) {
  unique_ptr<int> lp(new int(p));
  return lp;
}
```

We cannot copy a unique pointer but when the pointer is about to be destroyed

We can return a local pointer

Introduced with C++11

Weak pointers

- Weak pointers are smart pointers that do not control the lifetime of the object to which they point
  - Weak pointers point to object managed by shared pointers
    - Weak pointers are initialized from shared pointers
    - They do not "own" the object
  - Creating a weak pointer **does not change the counter** of the original shared pointer
    - To use a weak pointer, we must be sure it is still valid
    - We must use the functions **expired** and **lock**

Main operations

Smart pointers are **templates**. We must specify the type to which they point

Type	Main characteristics
weak_ptr<T> w;	Define a weak pointer w that can point to an object of type T. Initialize w to nullptr.
weak_ptr<T> w(sp);	Make the weak pointer w pointing to the same object the shared pointer sp is pointing to.
w = p;	Assign to the weak pointer w, the weak pointer or shared pointer p.
w.reset()	Make w null.
w.use_count()	Returns the number of shared pointers that points to the same object pointed by w.
w.expired()	Retuns true if w.use_count() is zero: false, otherwise.
w.lock()	If the counter is zero, it returns a null shared pointer. Otherwise, it returns a shared pointer to the object pointed by the weak pointer w.

Example

```
struct person{
  string name;
  person(string n):name(n){}
};
```

```
shared_ptr<person> p1 = make_shared<person>("Jack");
shared_ptr<person> p2;
shared_ptr<person> p3;

p2 = p1;
weak_ptr<person> wp(p1);

if (p3 = wp.lock()) {
  cout << p3->name << endl;     //  Jack
}

p1.reset(new person("rose"));
p2.reset();
p3.reset();

if (wp.expired()) {
  cout << "Pointer KO !" << endl;
}  else {
  cout << "Pointer OK: " << p3->name << endl;
}
```

Counter: 1

Counter: 2

Get the shared pointer underlying wp

Counter: 3

Counter: 1

Counter: 0

Red line: wp.expired is true
No red line: wp.expired is false

In any case wp cannot be used to access directly the resource; this must be done through lock() and a shared pointer

Circular dependency

- Weak pointers can be used to break circular dependency of shared pointers
  - Example
    - A→B using a shared pointer
    - B→A using a shared pointer
    - When A and B go out of scope they are not deleted, because the counters of the two pointers are not equal to zero
  - In circular dependency, we must use a weak pointer

Example

```
struct person;

struct team{
  shared_ptr<person> myp;
  ~team(){ cout << "Team destructed."; }
};

struct person{
  shared_ptr<team> myt;
  ~person(){ cout << "Person destructed."; }
};

int main(){
  auto team1 = make_shared<team>();
  auto person1 = make_shared<person>();

  team1->myp = person1;
  person1->myt = team1;

  return 0;
}
```

A teams points to a person

A person points to a team

Objects team1 and person1 refer each other

The counters of the two shared pointers are equal to one.
The two destructors are not called.
There is a memory leak

Example

```
struct person;

struct team{
  shared_ptr<person> myp;
  ~team(){ cout << "Team destructed."; }
};

struct person{
  weak_ptr<team> myt;
  ~person(){ cout << "Person destructed."; }
};

int main(){
  auto team1 = make_shared<team>();
  auto person1 = make_shared<person>();

  team1->myp = person1;
  person1->myt = team1;

  return 0;
}
```

A teams points to a person

A person points to a team

Objects team1 and person1 refer each other

The counter of myt is zero.
Thus person1 can be deleted.
Thus, the counter of myp becomes zero.
Also team1 can be deleted.