Memory Management

Pointer Casting, C++ Memory, Allocation, Deallocation

SoftUni Foundation

BY NC SA

Georgi Georgiev

A guy that knows C++

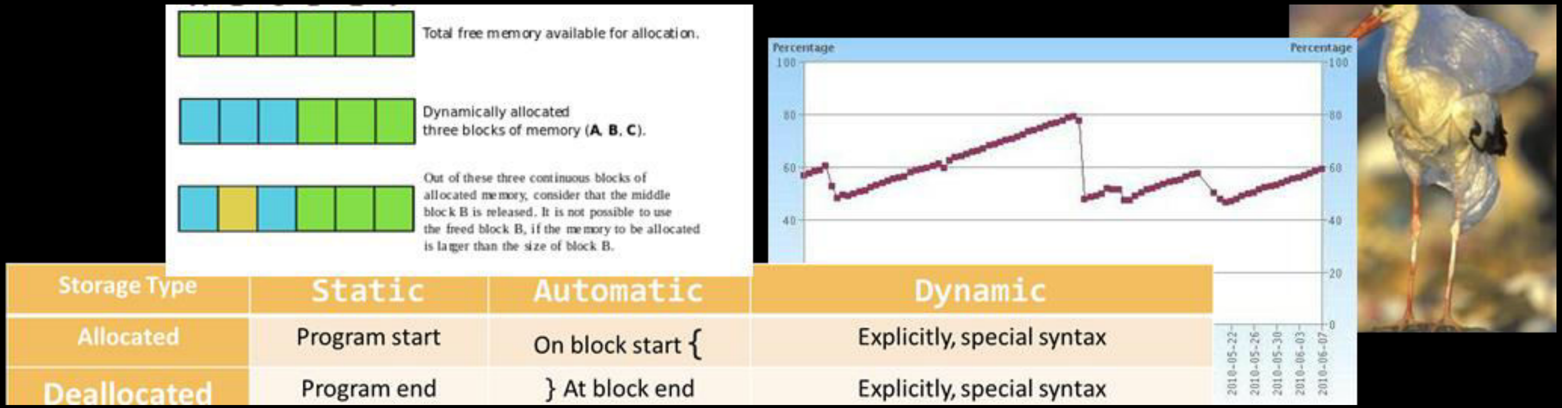


Table of Contents

Questions

SoftUni Foundation

sli.do

#cpp-softuni

Function Pointers

Accessing Functions Through Variables

Function Pointers

- Pointers (and references) can point to functions
  - `FuncReturnType (*name)(`*`function parameter types`*`)`
- Assign with name of a matching function
  - Use instead of function name

```cpp
vector<string> split(string s, char sep) {
  vector<string> strings;
  ...
  return strings;
}
```

```cpp
vector<string> (*p)(string, char);
p = &split; // this also works: p = split;

p("hello world", ' '); // returns { "hello", "world" }
```

Function Pointers

LIVE DEMO

Exercise 1: Function Pointer Applications

- "Normal" parameters – pass different data
- Function pointer parameters – pass different behavior
  - Called "callbacks"
- Exercise: function to filter a `vector<int>`
  - Accept pointer to function – to decide which numbers to filter
  - Filter numbers > 3
  - Filter even numbers
  - Filter negative numbers

Casting Pointers

Pointer to TypeA -> Pointer to TypeB

SoftUni Foundation

The void Pointer (void*)

- Just an address in memory
  - **void*** can point to anything
  - Other pointers implicitly cast to **void***
- No type information
  - Cannot reference/dereference
  - No pointer arithmetic

```
int number = 42;
char cStr[] = "hello";
char* otherCStr = "world";

void* p;
p = &number;
p = cStr;
p = otherCStr;
cout << p; // prints address

p++; // compilation error
cout << *p; //compilation error
```

void*

LIVE DEMO

Casting Pointers

- All pointers can be casted
  - Specific-> general = implicit cast (e.g. **`int*`** -> **`void*`**)
  - General -> specific = requires explicit cast (e.g. **`void*`** -> **`int*`**)
- C-Style casting can be used, NOT recommended

```
char letter = 'A';
void* voidPtr = &letter;

char* cStyleCastPtr = (char*)voidPtr;
```

Quick Quiz TIME:

SoftUni Foundation

- You are John Snow. What will the following code print?

```
char letter1 = 'a', letter2 = 'b', letter3 = 'c', letter4 = 'd';
int* letter4Ptr = (int*)&letter1;

*letter4Ptr = 842281524;

cout << letter1 << letter2 << letter3 << letter4 << endl;
```

a) It will print **abcd**

b) Nothing, it will cause a compilation error

c) It will summon demons

d) You don't know

C++ PITFALL: UNCHECKED ACCESS TO C-STYLE CASTED POINTER MEMORY

C-Style pointer casting doesn't check type, so you can change to any type of pointer. But that's dangerous.

E.g. an `int*` on a system where `int` is **4** bytes, assigned to the address of a **1**-byte `char`, will access **4** bytes, **3** of which are not guaranteed to be part of your program.

C++ Pointer Casting

- `static_cast<T>` – compile-time type checking

```
char letter = 'A';
void* voidPtr = &letter;
char* p1 = static_cast<char*>(voidPtr); // no checks for void*
int* p2 = static_cast<int*>(p1); // compilation error
```

- `dynamic_cast<T>` (classes) – runtime checks, `nullptr` if failure
- `const_cast<T>` – changes `const`-ness
- `reinterpret_cast` – no checks, just gives wanted type
  - *Avoid like the plague, unless you're the plague doctor*

Casting Pointers

LIVE DEMO

C++ Memory Types

Automatic, Dynamic, Static

Memory & Programs

- Memory has a pattern of usage
  - Request memory – "Allocation"
  - Use memory
  - Release memory when done – "Deallocation"
- C++ storage types for variables
  - Describe how memory is handled, i.e. the "lifetime" of objects

C++ Storage Types

SoftUni Foundation

- Static – marked with `static`
- Automatic – NO ~~`static`~~, ~~`extern`~~, ~~`thread_local`~~, ~~`register`~~, ~~`mutable`~~
  - Locals, parameters, etc.
- Dynamic – allocated/deallocated by special syntax

Storage Type	Static	Automatic	Dynamic
Allocated	Program start	On block start {	Explicitly, special syntax
Deallocated	Program end	} At block end	Explicitly, special syntax
Lifetime	Entire program	Scope	From allocation to deallocation

Automatic Storage Example

SoftUni Foundation

- Until now, all our non-`static` variables were automatic

```
void allocateLargeAutoVector() {
  vector<int> autoVector;
  for (size_t i = 0; i < 1000000; i++) autoVector.push_back(i);
}
```

autoVector lifetime

```
int main() {
  for autoVar = 0;
  for (size_t i = 0; i < 1000000; i++)
    int autoVarLoop = a * b;
    autoVar += autoVarLoop;
  }

  allocateLargeAutoVector();
  return 0;
}
```

autoVarLoop lifetime

autoVar lifetime

Automatic Storage

LIVE DEMO

Automatic Storage Limitations

SoftUni Foundation

- Bad (almost always) to **`return`** pointer/reference to automatic locals
  - Copies usually necessary (before C++11)

```
vector<double> getPrecomputedSquareRoots() {
  vector<double> roots;
  for (size_t i = 0; i < 1000000; i++) roots.push_back(sqrt(i));
  return roots;
}
```

Copying 1M elements (C++11 optimizes this)

- Automatic *usually* allocated on program stack
  - Faster, but very limited memory
  - **`int arr[1000000];`** causes runtime error on most systems

Dynamic Memory

User-Controlled Allocation & Deallocation

Dynamic Memory Allocation

SoftUni Foundation

- The **operator `new`** manually allocates memory
  - Returns typed pointer to allocated memory
- **`new` *`T(constructor params)`*** – single object
- **`new` *`T[size] {initializer list}`*** – array

```cpp
int* arr = new int[]{ 42, 13, 255 };
cout << arr[0] << " " << arr[1] << " " << arr[2];
```

```cpp
Person* person = new Person("John", 20);
cout << person->name; // prints "John"

Person* people = new Person[3]; // compilation error
```

```cpp
class Person {
public:
  string name; int age;
  Person(string name, int age)
  : name(name)
  , age(age) {}
};
```

`operator new`

LIVE DEMO

Dynamic Memory Deallocation

- The **operator delete** deallocates **new**-allocated memory
  - E.g. if `T* p = new T(); T* arr = new T[size];` then `delete p;` but `delete[] arr;`
- Should **delete** when done using memory
  - Accessing is undefined after deletion

```
int* arr = new int[]{ 42, 13, 255 };
cout << arr[0] << " " << arr[1] << " " << arr[2];
delete[] arr;
```

```
Person* p = new Person("John", 20);
delete p;
cout << p->name; // undefined behavior
```

- *Good practice: set pointer to* **nullptr** *after* **delete**

Managing Memory – new & delete

SoftUni Foundation

- Release any **new**-allocated memory when not using it anymore
  - With **delete/delete[]**

```
double* roots = getRoots(100);

int numbers; cin >> numbers;
for (int i = 0; i < numbers; i++) {
  int number; cin >> number;
  cout << roots[number];
}
delete[] roots;
```

```
double* getRoots(int to) {
  double* roots = new double[to + 1];
  for (size_t i = 0; i <= to; i++) {
    roots[i] = sqrt(i);
  }
  return roots;
}
```

- Avoid **delete**-ing **nullptr** – *the standard is a bit fuzzy on it*

Quick Quiz TIME:

SoftUni Foundation

- What will the following code do?

```
int* numbers = new int[3] { 1, 2, 3 };
int* otherPtr = numbers;
delete[] numbers;
delete[] otherPtr;
```

a) It will allocate then deallocate memory and exit successfully

b) There will be a runtime error

c) There will be a compilation error

d) Behavior is undefined

C++ PITFALL: DELETING MEMORY MORE THAN ONCE

A single **new** needs a single **delete**.

delete the memory, NOT the pointer.

After memory is **delete**d, any following **delete**s access memory you do not own – undefined behavior

Memory Leaks

- If no **delete**, we get a memory leak
  - Program keeping unused memory
  - System can't "recycle" memory

```
int numbers; cin >> numbers;
for (int i = 0; i < numbers; i++) {
  int number; cin >> number;
  cout << getRoots(100)[number]; // memory leak
}
```

```
double* getRoots(int to) {
  double* roots = new double[to + 1];
  ...
  return roots;
}
```

- Leaks are rarely obvious
  - Minimize **new** usage, think about **delete** for every **new**

Releasing Unused Memory

LIVE DEMO

Exercise 2: Print Largest Sum Array

- You are given integer **N** – number of arrays
  - Each array entered as integer **L** followed by exactly **L** integers
- Print the one with the largest sum
- You are NOT allowed to use any STL container
  - i.e. you must use **`new`** and **`delete`** for the arrays
  - Avoid memory leaks!

```
/*test.000.001.in.txt*/
3
4   1  5  2  11
2   10  42
5   1  -2  3  0  -3
```

```
/*test.000.001.out.txt*/
10 42
```

Quick Quiz TIME:

SoftUni Foundation

- What will the following code do?

```cpp
auto people = getPeople();
cout << people->at(0)->getName();
delete people;
```

```cpp
vector<Person*>* getPeople() {
  auto people = new vector<Person*>();
  people->push_back(new Person("Ben Dover", 42));
  people->push_back(new Person("Ary O'usure", 25));
  return people;
}
```

a) It will print **Ben Dover**, and leak memory

b) It will print **Ben Dover**, without leaking memory

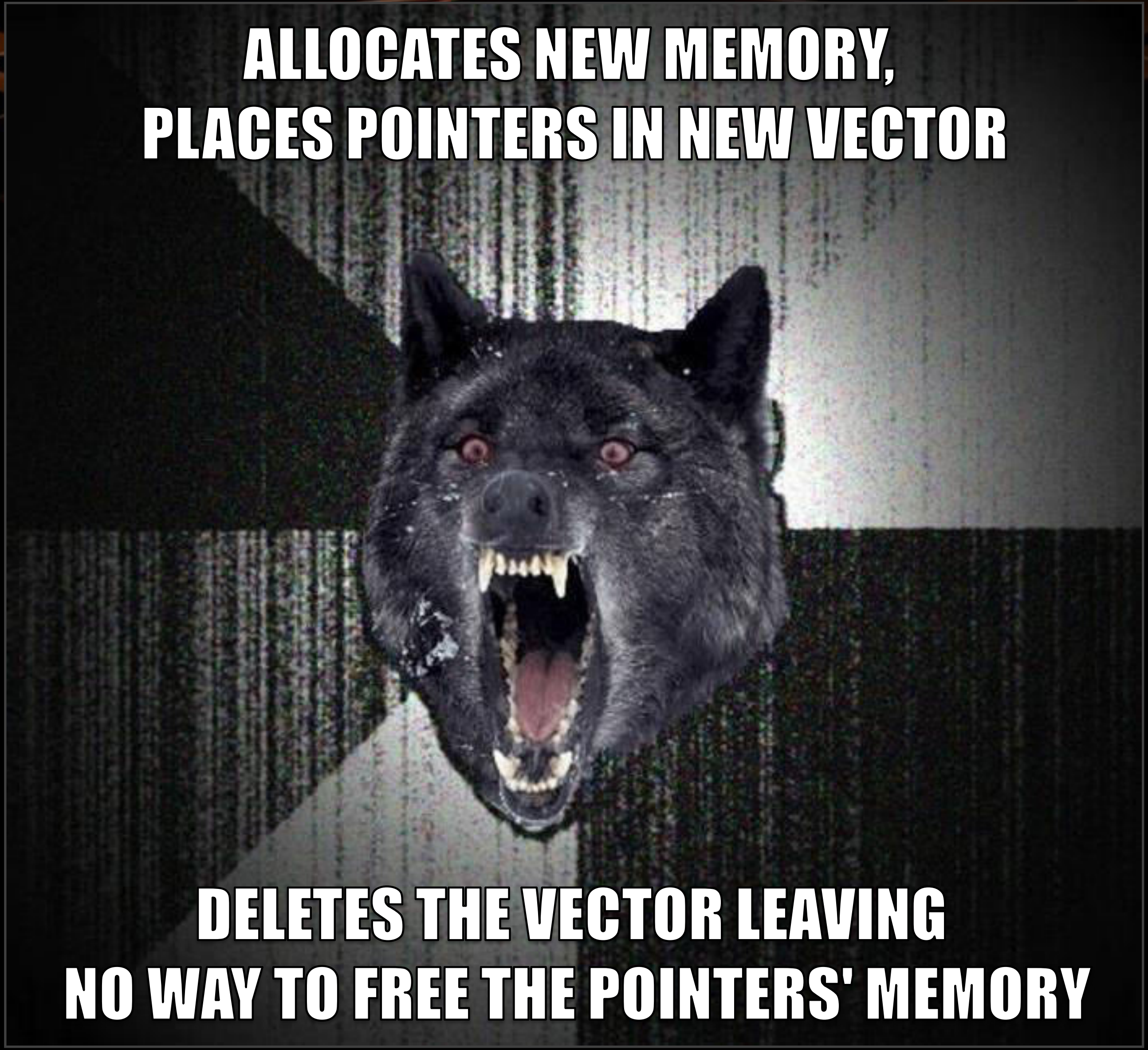
c) There will be a runtime error

d) There will be a compilation error

C++ PITFALL: DELETING POINTER CONTAINER, BEFORE DELETING POINTERS INSIDE

Every **`new`** needs a **`delete`**.

If you **`delete`** a container that has **`new`**-initialized pointers before **`delete`**-ing the pointers, you have no way of freeing their memory later.



What About C-Style Memory Functions?

- C has `malloc()`, `calloc()`, `realloc()`, for allocation
- And `free()` for deallocation
- MAJOR difference with `new`/`delete`:
  - C versions don't call constructors/destructors
  - Just request/release bytes from/to system
  - i.e. no guarantee of valid objects allocation, just memory size
  - i.e. no guarantee of complex object deallocation

C++ Smart Pointers

Avoiding the `new`/`delete` hassle

Smart Pointers

- Similar operations to "raw" **`T*`** pointers, plus:
  - Automate some part of memory management
  - **`reset(T*)`** – changes pointer, **`T* get()`** returns raw pointer
  - **`operator bool`**, has **`true`** value if non-**`nullptr`**

```
unique_ptr<Person> personPtr(new Person("John", 20));
cout << personPtr->getName() << endl;
// no need for delete, unique_ptr clears memory when it goes out of scope
```

```
unique_ptr<Person> personPtr = make_unique<Person>("John", 20); // C++14
cout << personPtr->getName() << endl;
// no need for delete, unique_ptr clears memory when it goes out of scope
```

Unique Pointer – unique_ptr<T>

SoftUni Foundation

- Deallocates memory when going out of scope
- Cannot copy the **`unique_ptr`** object – compilation error

```
unique_ptr<Person> personPtr(new Person("John", 20));
unique_ptr<Person> copy = personPtr; // compilation error
```

- Use when you want exactly **1** pointer to the object
  - Can pass around reference to the pointer
  - Prevents creating accidental copies

unique_ptr

LIVE DEMO

Shared Pointer – shared_ptr<T>

SoftUni Foundation

- Tracks number of copy pointers
  - Deallocates when last goes out of scope
  - Construct with allocated memory, or with **make_shared<T>**

```
void f() {
  shared_ptr<Person> longerCopy;
  if (...) {
    shared_ptr<Person> person(new Person("James", 23));
    shared_ptr<Person> copy = person;
    longerCopy = person;
  }
  cout << longerCopy->getName() << endl;
}
```

1 pointer

2 pointers

3 pointers

person & copy out of scope, 1 pointer remains

last pointer out of scope (longerCopy), memory deallocated

shared_ptr

LIVE DEMO

Summary

SoftUni Foundation

- Pointers can point to and call functions
- Pointers implicitly cast to "more general" types
  - Explicitly cast to "more specific" types, e.g. with **`static_cast`**,
- Automatic memory is allocated and deallocated in a scope
- Dynamic memory is managed manually
  - new allocates, requires delete to deallocate
- **`unique_ptr`** and **`shared_ptr`** do deletion automatically

Memory Management

SoftUni Foundation

xssoftware

SmartIT

NETPEAK
SEO and PPC for Business

SUPERHOSTING®.BG

INDEAVR
Serving the high achievers

Questions?

telenor

SOFTWARE GROUP

INFRAGISTICS™
DESIGN / DEVELOP / EXPERIENCE