

# Memory Management

Pointer Casting, C++ Memory, Allocation, Deallocation



**SoftUni Team**  
Technical Trainers



**SoftUni**



Software University

<https://softuni.bg>

# Table of Contents

1. Function Pointers
2. Pointer Casting
3. Memory Types
  - Automatic
  - Dynamic
4. Dynamic Memory
  - Allocation and Deallocation
5. Smart Pointers





sli.do

**#cpp-advanced**




# Function Pointers

Accessing Functions Through Variables

# Function Pointers

- Pointers (and references) can point to functions
- Assign with name of a matching function
  - Use instead of function name



```
vector<string> split(string s, char sep) {  
    vector<string> strings;  
    ...  
    return strings;  
// function return type (*name) (function parameter)  
}
```

```
vector<string> (*p)(string, char);  
p = &split; // this also works: p = split;  
p("hello world", ' '); // returns { "hello", "world" }
```

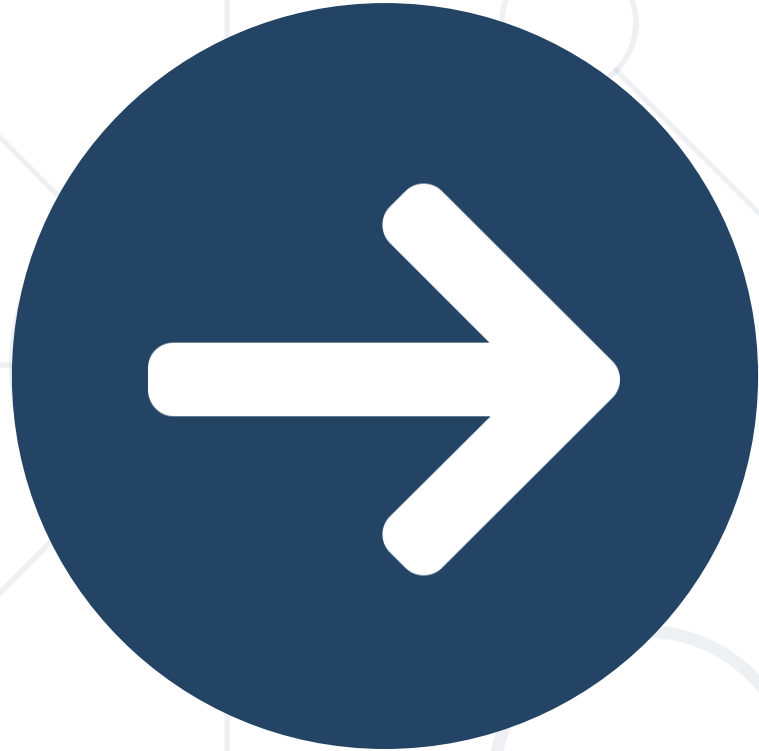


# Function Pointers

LIVE DEMO

# Problem 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



**Pointer Casting**



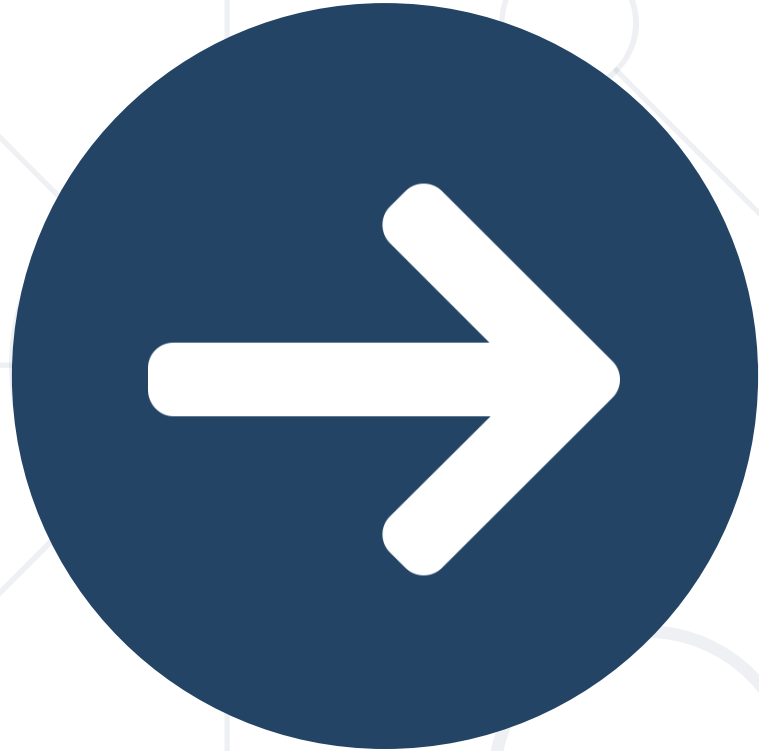
# The void Pointer (void\*)

- 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
```




# The void Pointer (void\*)

LIVE DEMO

# Pointer Casting

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



```
char letter = 'A';  
void* voidPtr = &letter;  
  
char* cStyleCastPtr = (char*)voidPtr;
```

- 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** does not check type, so you can change to any type of pointer. But this is dangerous.

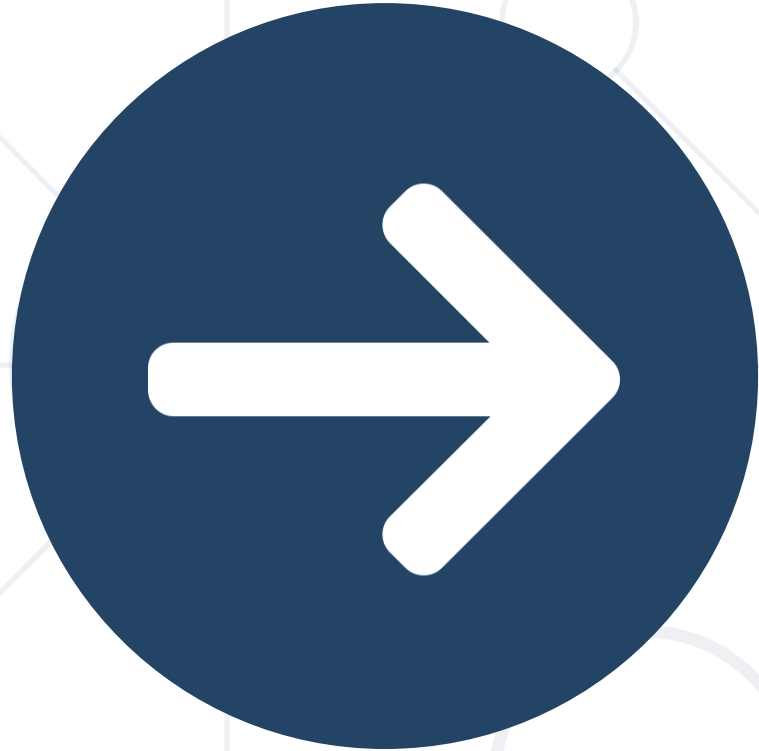
E. g. an **int\*** on a system where **int** is **4** bytes, assigned to the address of a **1-byte char**, will access **4** bytes, **3** of which are not guaranteed to be part of your program.



- **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>** – runtime checks, **nullptr** if failure
- **const\_cast<T>** – changes **const**-ness
- **reinterpret\_cast** – no checks, just gives wanted type



# Pointer Casting

LIVE DEMO




# 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 ("lifetime" of objects)

- Static – marked with **static**
- Automatic
  - Locals, parameters, etc.
- Dynamic – allocated/deallocated by special syntax

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

# Automatic Storage Example

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

autoVar lifetime

autoVarLoop lifetime

```
    allocateLargeAutoVector();  
    return 0;  
}
```



# Automatic Storage

LIVE DEMO

- It is bad to return pointer/reference to automatic locals

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

- 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 and Deallocation

# Dynamic Memory Allocation

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



# Dynamic Memory Allocation - Example

```
int* arr = new int[] { 42, 13, 255 };
cout << arr[0] << " " << arr[1] << " " << arr[2];
Person* person = new Person("John", 20);
cout << person->name; // prints "John"

Person* people = new Person[3]; // compilation error
class Person {
public:
    string name; int age;
    Person(string name, int age) : name(name), age(age) {}
};
```





**Operator new**

LIVE DEMO

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

```
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
```

# Managing Memory – new & delete

- 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**



# Dynamic Memory

LIVE DEMO


- 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

# 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
}
```

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

# Problem 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 input*/
```

```
3
4  1 5 2 11
2  10 42
5  1 -2 3 0 -3
```

```
/*test output*/
```

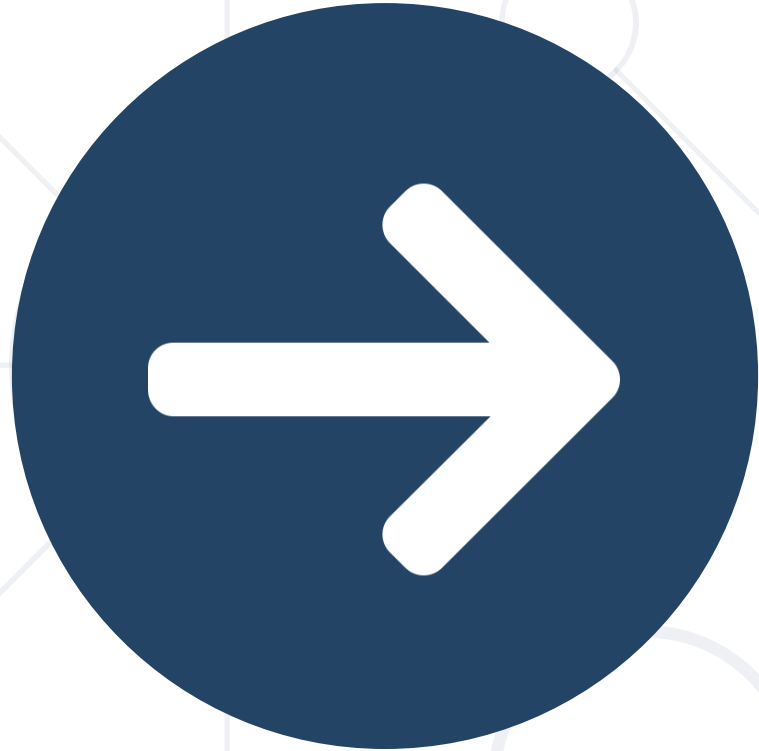
```
10 42
```

■ What will the following code do?

- 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

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





**Smart Pointers**

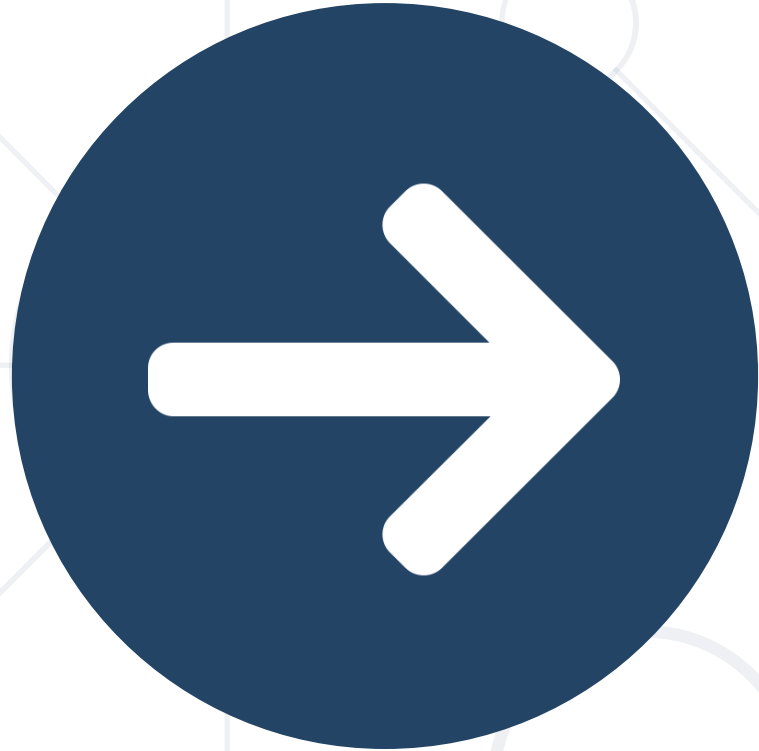
- Similar operations to "raw" **T\*** pointers, plus:
  - Automate some part of memory management
  - **reset(T\*)** – changes pointer, **T\* get()** returns raw pointer
  - **operator bool**, has **true** value if non-**nullptr**

```
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_ptr<T>`
- 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 one pointer to the object
  - Can pass around reference to the pointer
  - Prevents creating accidental copies

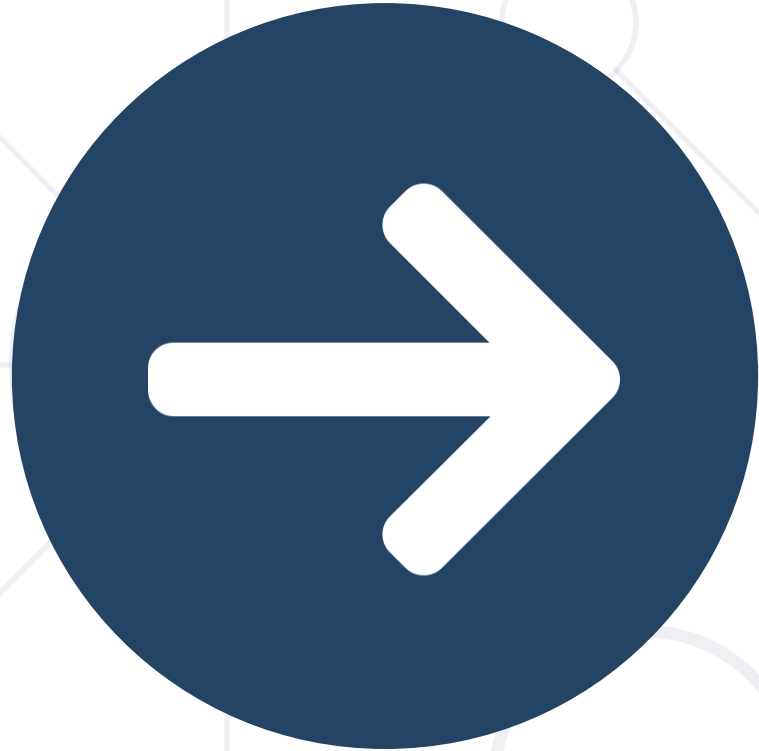


# Unique Pointers

LIVE DEMO

- **shared\_ptr<T>**
- 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;  
}
```



# Shared Pointers

LIVE DEMO

- 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



# Questions?





# SoftUni Diamond Partners

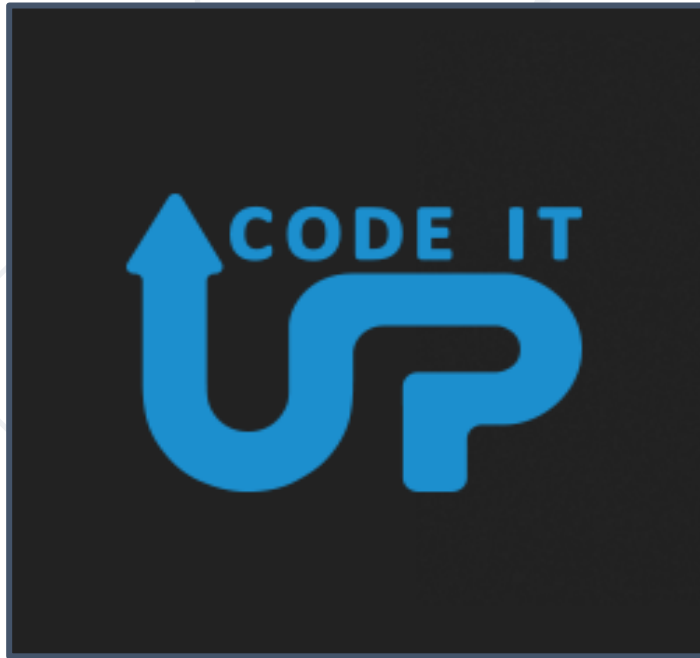


**SCHWARZ**



**SUPER  
HOSTING  
.BG**





**VIRTUAL RACING SCHOOL**



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)

