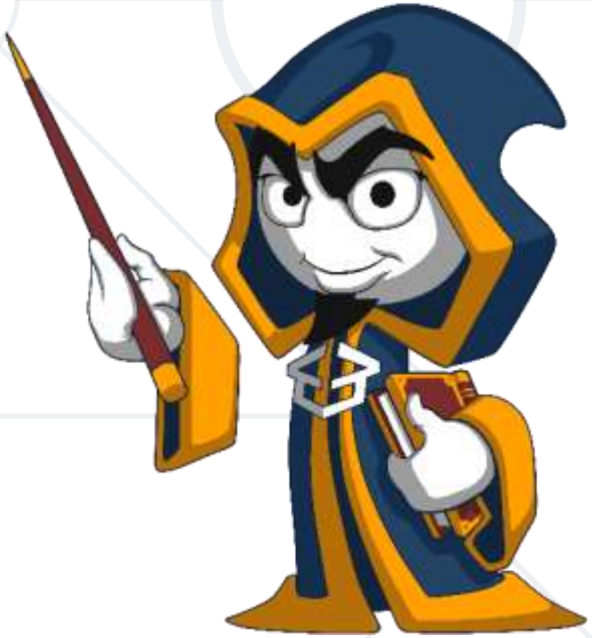
Memory Management

Pointer Casting, Memory, Allocation, Deallocation

SoftUni Team

Technical Trainers

Software University



SoftUni

Software University

https://softuni.bg

Have a Question?

sli.do

#cpp-advanced

Table of Contents

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" }
```

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

```cpp
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
```

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;
```

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

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)

C++ Storage Types

- **Static** – marked with `static`
- **Automatic**
  - Locals, parameters
- **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

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

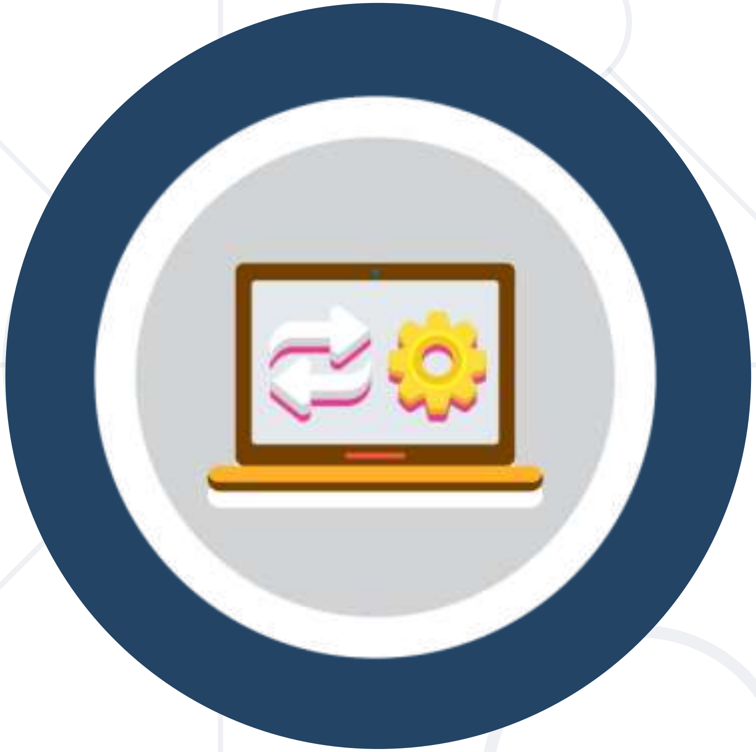
```
void allocateLargeAutoVector()
{
  vector<int> autoVector;
  for (size_t i = 0; i < 1000000; i++) autoVector.push_back(i);
}
int main()
{
  int autoVar = 0;
  for (size_t i = 0; i < 1000000; i++)
  {
    int autoVarLoop = a * b;
    autoVar += autoVarLoop;
  }

  allocateLargeAutoVector();
  return 0;
}
```

Automatic Storage Limitations

- 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) {}
};
```

Dynamic Memory Deallocation

- The `operator delete` deallocates `new`-allocated memory
  - 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
```

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**

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**

Smart Pointers

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>`
- 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**

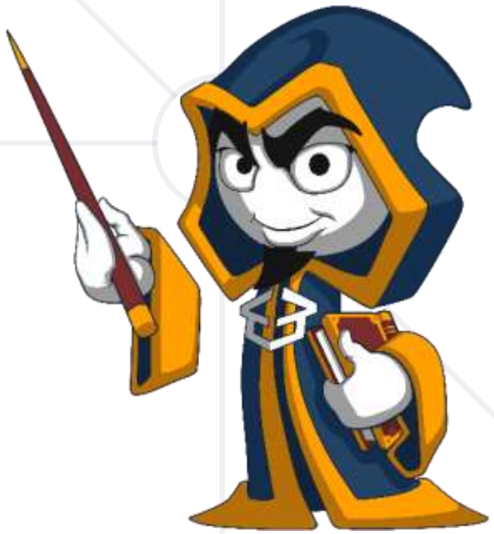
Shared Pointer

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

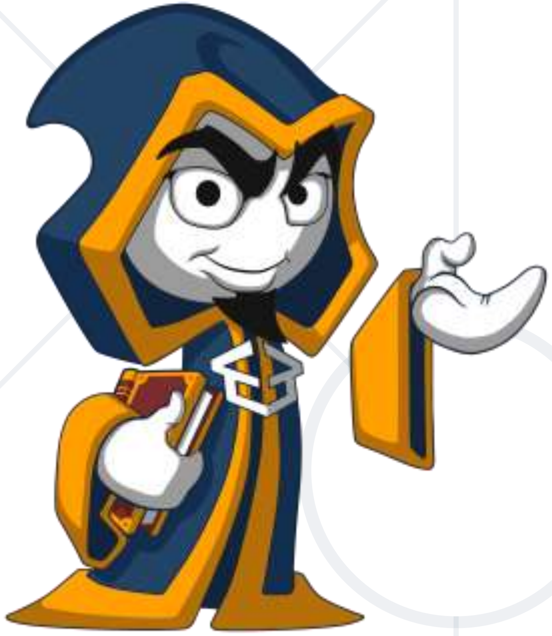
```
void f()
{
  shared_ptr<Person> longerCopy;
  if (...)
  {
    shared_ptr<Person> person(new Person("James", 23));
    shared_ptr<Person> copy = person;
    longerCopy = person;
   }
  cout << longerCopy->getName() << endl;
}
```

Summary

- 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

- Software University
- SoftUni Creative
- SoftUni Digital
- SoftUni Foundation
- SoftUni Kids
- СофтУни БУДИТЕЛ
- Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

Trainings @ Software University (SoftUni)

Software University
- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

Software University

License