Structures

CSE 232 - Dr. Josh Nahum

Reading:

Section 2.1 and Section 2.2

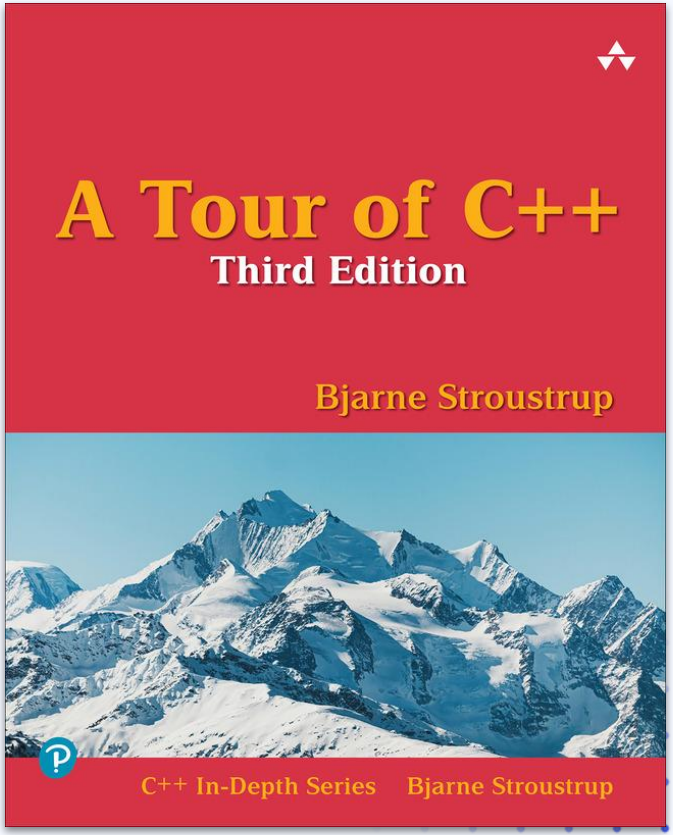

Table of contents

OO

& and Functions

Argument Passing

Pass By Value

Example

```
int Count(string s);
```

Effect

The argument is copied, and thus no changes to it affect the original

Pass By Reference

Example

```
int Count(string & s);
```

Effect

The parameter is a reference to the argument, thus changes to the parameter affect the argument

When to Pass By Value versus Reference

Pass By Value

- When you need a copy
- When copying is cheap (fundamental, fixed-width, small types, like int or pointers)

Pass By Reference

- When you want to avoid copying
- When copying is expensive (large type, especially user-defined types or containers)
- When you want to be able to change the argument

Common Questions

When should I make a function parameter const?

Const for a variable declaration has the same meaning as for a parameter declaration, if the variable shouldn't change after initialization, mark it const.

When should I make a parameter a pointer?

Very rarely. Generally if you are considering making a parameter a pointer, a reference is usually the better option. That said, there are circumstances when functions that take a pointer are needed (generally involving arrays, more on this later).

01

Access Operators

“-> (the arrow operator) is syntactic sugar, meaning it isn't necessary, but is useful for writing legible code.”

— me

Dot and Arrow Operators

```
string s{"CSE 232"};
cout << s.size();
// Dot operator allows access
// to the members of the string

string * s_ptr = &s;
cout << s_ptr->size();
// Arrow operator does the same
// thing, but dereferences the
// pointer first.
```

```
cout << (*s_ptr).size();
// This does the same thing as
// the previous arrow operator.
cout << *s_ptr.size();
// This is a compile time error
// The . (dot) has a high operator
// precedence than *, so it means
// the same as this
cout << *(s_ptr.size());
// which is a compile time error
// because pointers don't have members
```

02

Dynamic Memory

The new operator

```
int * x = new int{0};
// new returns a pointer to
// dynamically allocated memory
// this object's lives until delete
// is called on the pointer

...

delete x;
```

```
std::string * words = new
  string[40]{"CSE", "232"};
// new can be used to dynamically
// allocate arrays too
// the initialization is optional,
// just like for regular variables

...

delete [] words;
// Note [] required to delete
// dynamically allocated arrays
```

Benefits of Dynamic Allocation

01 — **Runtime Size** — Static allocation requires the size of types to be known at compile time.

02 — **Scope** — A statically allocated variable's scope is determined by the declaration location.

03 — **Heap** — Dynamically allocated variables are allocated on the "heap" a memory location with different performance implicates than the "stack".

04 — **Control** — Advanced C++ software developers can use custom allocators to optimize for memory performance.

03

Live Coding

Attribution

Please ask questions via Piazza

Dr. Joshua Nahum
www.nahum.us
EB 3504

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**