Ch 3: Reference (data) Types

CSCI 330

Overview

Reference types store the memory address of an objects

Key topics:

- **Pointers**: explicit memory addresses
- **References (pass-by-reference)**: safe, alias-style access to original address
- ***this***: pointer to current object
- ***const***: read-only guarantees
- ***auto***: type deduction for cleaner code

Pointers

A pointer stores the memory address of another object.

- The type of object it points to
- The address in Memory

Declaring and using pointers (note location of *)

```cpp
#include <cstdio>

int main() {
    int value = 42;                 // Declare an integer
    int* ptr = &value;              // Pointer stores the address of value

    printf("Address of value: %p\n", &value);  // Print address directly
    printf("Value of ptr:    %p\n", ptr);      // Print pointer (should match address of value)
    printf("Dereferenced ptr: %d\n", *ptr);    // Dereference to access value (prints 42)

    *ptr = 99;                      // Modify the value via pointer
    printf("Updated value:    %d\n", value);   // Confirm it changed

    return 0;
}
```

```
cpp — -zsh — 62×9
% clang++ -std=c++17 -o pointerExample pointerExample.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp
% ./pointerexample
Address of value: 0x16ddbf208
Value of ptr:    0x16ddbf208
Dereferenced ptr: 42
Updated value:    99
davideve@MacBook-Pro-2024 ~/DevEnv/cpp
%
```

Addressing Variables

- Use & to get memory address of a variable
- Use * to define or dereference a pointer
- Common in system-level programming (OS APIs)

ch3 > pointerAddress.cpp > main()

```cpp
#include <cstdio>

int main(){
    int gettysburg{};
    printf("gettysburg: %d\n", gettysburg);
    int *gettysburg_address = &gettysburg;
    printf("&gettysburg: %p\n", gettysburg_address);
}
```

ch3 — -zsh — 62×11

```
% clang++ -std=c++17 -o pointerAddress pointerAddress.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% ./pointerAddress
gettysburg: 0
&gettysburg: 0x16bd1f1fc
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% ./pointerAddress
gettysburg: 0
&gettysburg: 0x16b63f1fc
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
%
```

Dereferencing Pointers

- Dereferencing operator (*) provides access to object that is referenced.
- Inverse of the address-of operator (&)
- Syntax Declaring a pointer w/ trailing *
- Dereferencing a pointer w/ leading *

ch3 > pointerAddress.cpp > main()

```cpp
#include <cstdio>

int main(){
    int gettysburg{};
    int *gettysburg_address = &gettysburg;
    printf("%d\n", *gettysburg_address); //read: prints 0
    *gettysburg_address = 17325;         //write
    printf("%d\n", *gettysburg_address); //read: prints 1732
}
```

ch3 — -zsh — 62×7

```
% clang++ -std=c++17 -o pointerAddress pointerAddress.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% ./pointerAddress
0
17325
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
%
```

The Member-of-Pointer (->) Operator

-> Performs two operations

- dereferences a pointer (returns stored value)
- accesses a member of the pointed-to object

ch3 > clockOfNow.cpp > main()

```cpp
#include <cstdio>

struct ClockOfTheLongNow {
    ClockOfTheLongNow() : year(2025) {}
    ClockOfTheLongNow(int y) { year = (set_year(y) ? y : 2025); }

    void add_year() { year++; }
    bool set_year(int y) { return (y < 2025) ? false : (year = y, true); }
    int get_year() const { return year; }

private:
    int year;
};

int main() {
    ClockOfTheLongNow clock;
    ClockOfTheLongNow* clock_ptr = &clock;
    clock_ptr->set_year(2025);
    printf("Address of clock: %p\n", clock_ptr);
    printf("Value of clock's year: %d", clock_ptr->get_year());
}
```

ch3 — -zsh — 62×8

```
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% clang++ -std=c++17 -o clockOfNow clockOfNow.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% ./clockOfNow
Address of clock: 0x16d0ab20c
Value of clock's year: 2025%
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
%
```

Pointers and Arrays

- Pointers store the location of a single object
- Arrays store the location and length of contiguous objects
- Arrays can decay into pointers, losing their length information

Example:

```
int key_to_the_universe[]{ 3, 6, 9 };
int* key_ptr = key_to_the_universe; // Points to 3
```

Pointer and Arrays:

ch3 > arrayDecay.cpp > main()

```cpp
#include <cstdio>

struct College{
    char name[256];
};
// Function accepts a pointer to a College (decayed from array)
void print_name(College* college_ptr){
    printf("%s College\n", college_ptr->name);
}
int main(){
    //Array of College structs
    College best_colleges[] = {"MCLA", "Amherst", "Beloit"};

    //Array decays to pointer to first element when passed to function
    print_name(best_colleges);
}

```

OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTGRESQL QUERY RESULTS

```
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 % clang++ -std=c++17 -o arrayDecay arrayDecay.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 % ./arrayDecay
MCLA College
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 %
```

Standard Practice (idiom) for Handling Decay

- Common problem: When passing arrays to functions, array decay removes size info

- Idiom/best practice: Pass two arguments together:
  - A pointer to the first array element
  - The number of elements in the array

```cpp
#include <cstdio>
struct College{
    char name[256];
};

void print_names(College* colleges, size_t n_colleges){
    for (size_t i = 0; i < n_colleges; i++){
        printf("%s College\n", colleges[i].name);
    }
}

int main(){
    College oxford[] = { "MCLA", "Amherst", "Beloit"};
    print_names(oxford, sizeof(oxford) / sizeof(College));
}
```

OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTGRESQL QUERY RESULTS

```
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 % clang++ -std=c++17 -o arrayDecayIdiom arrayDecayIdiom.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 % ./arrayDecayIdiom
MCLA College
Amherst College
Beloit College
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 %
```

Pointer Arithmetic

Accessing the *n-th* element's address in an array:

- Use bracket indexing + address-of operator (line 9)
- Use pointer arithmetic (line 12)

Both generate the same result

ch3 > pointerArithmetic.cpp > main()

```cpp
1   #include <cstdio>
2
3   struct College { char name[256]; };
4
5   int main() {
6     College oxford[] = { "MCLA", "Amherst", "Beloit" };
7
8     // Technique 1: Bracket indexing with address-of
9     College* ptr1 = &oxford[2];
10
11    // Technique 2: Pointer arithmetic
12    College* ptr2 = oxford + 2;
13
14    // Both point to "Kellogg"
15    printf("ptr1 points to: %s\n", ptr1->name);
16    printf("ptr2 points to: %s\n", ptr2->name);
17  }
```

OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTGRESQL QUERY RESULTS

```
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 % clang++ -std=c++17 -o pointerArithmetic pointerArithmetic.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 % ./pointerArithmetic
ptr1 points to: Beloit
ptr2 points to: Beloit
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
 %
```

Pointers are Dangerous

- Power vs Risk
  - Pointers provide low-level direct memory access
  - Enable powerful manipulation (essential in systems programming)
  - No built-in bounds checking (makes them risky)
- Buffer Overflows
  - Arrays & pointers allow out-of-bounds access via [] or arithmetic
  - Compiler doesn’t stop illegal memory access
  - Leads to undefined behavior: crashes, corruption, or worse

Buffer Overflow Example

ch3 > bufferOverflowExample.cpp > main()

```cpp
int main(){
    char upper[] = "ABC?E";
    char* upper_ptr = upper; //Equivalent: &upper[0]

    lower[3]='d';  //lower now contains a b c d e
    upper_ptr[3] = 'D'; // upper now contains A B C D E

    char letter_d = lower[3]; //letter_d = 'd'
    char letter_D = upper[3]; //letter_D = 'D'
    printf("lower: %s\nupper: %s", lower, upper);

    lower[7] = 'g';  // Super bad buffer overflow
}
```

ch3 — -zsh — 62×8

```
pointerArithmetic
pointerArithmetic.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% ./bufferOverflowExample
lower: abcde
upper: ABCDE%
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
%
```

OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTGRESQL QUERY RESULTS zsh - ch3

```
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% clang++ -std=c++17 -o bufferOverflowExample bufferoverflowExample.cpp
bufferoverflowExample.cpp:15:5: warning: array index 7 is past the end of the array (which contains 6 elements)
      [-Warray-bounds]
    lower[7] = 'g';  // Super bad buffer overflow
    ^     ~
bufferoverflowExample.cpp:4:5: note: array 'lower' declared here
    char lower[] = "abc?e";
    ^
1 warning generated.
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
%
```

Connection between Brackets and Pointer Arithmetic

- array[index] is equivalent to *(array + index)
- The bracket syntax is pointer arithmetic (equivalent)

```
lower[] = ‘abcde’;
*(lower + 7) = ‘g’;
lower[7] = ‘g’;
```

Remember:

- Arrays have fixed bounds, but pointer arithmetic doesn’t enforce them!
- Accessing *(lower + 7) is undefined behavior
- bugs may not appear where bad write occurs

Void and std::byte Pointers

- Use void* when type information is irrelevant
  - type-erased pointers
  - cannot be dereferenced or used with pointer arithmetic
  - common in C APIs and generic containers
- Use std::byte* for raw memory access
  - introduced in C++17
  - Designed for byte-wise manipulation
  - Safe alternative to char* when dealing with raw memory

nullptr and Boolean Expressions

nullptr is a special literal indicating a pointer does not point to anything

- Commonly used to signal errors or out-of-memory conditions
- pointers implicitly convert to bool:
  - nullptr = false
  - non-null pointers = true

Best practice:

- Functions that return pointers use **nullptr** to indicate failure (memory allocation functions)

References

References are safer, cleaner pointers

- Declared using &: Type& name
- Cannot be null
- Cannot be reseated (bound once)

Cleaner syntax than pointers

- no need for * or ->
- used like actual object

h3 > COLN.cpp > main()

```cpp
#include <cstdio>

struct ClockOfTheLongNow {
  ClockOfTheLongNow(int year_in) {…
  ClockOfTheLongNow() {…
  void add_year() {…
  bool set_year(int new_year) {…
  int get_year() {…

  private:
  int year;
};

void add_year(ClockOfTheLongNow& clock) {
  clock.set_year(clock.get_year() + 1); // No deref operator needed
}

int main() {
  ClockOfTheLongNow clock;
  printf("The year is %d.\n", clock.get_year());
  add_year(clock); // Clock is implicitly passed by reference!
  printf("The year is %d.\n", clock.get_year());
}
```

ch3 — -zsh — 62×8

```
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% clang++ -std=c++17 -o coln COLN.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
% ./coln
The year is 2019.
The year is 2020.
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
%
```

Usage of Pointers and References

- Pointers and references are often interchangeable
- User pointers when
  - You need to change what the variable refers to
  - the data structures (e.g., linked lists) require reassignment
- References:
  - Cannot be reseated
  - Should not be assigned nullptr
  - Not suitable when reassignment is needed.

Forward-Linked Lists: Canonical Pointer-Based Data Structure

- A forward-linked list is a sequence of elements, each pointing to the next
- The last element points to **nullptr**
- Elements can be stored **non-contiguously** in memory
- Insertions are efficient – only pointer values are updated

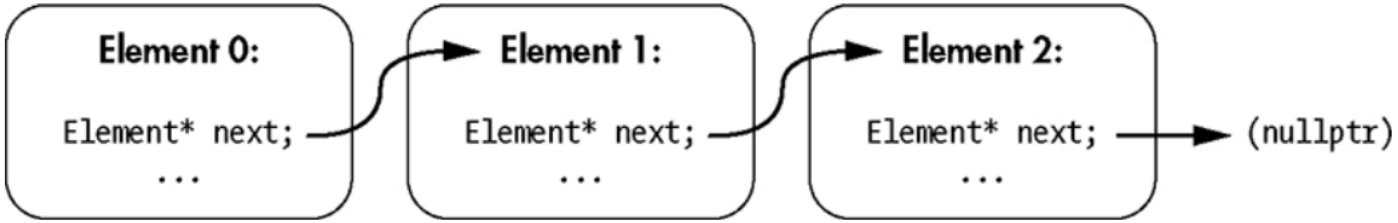

Figure 3-1: A linked list

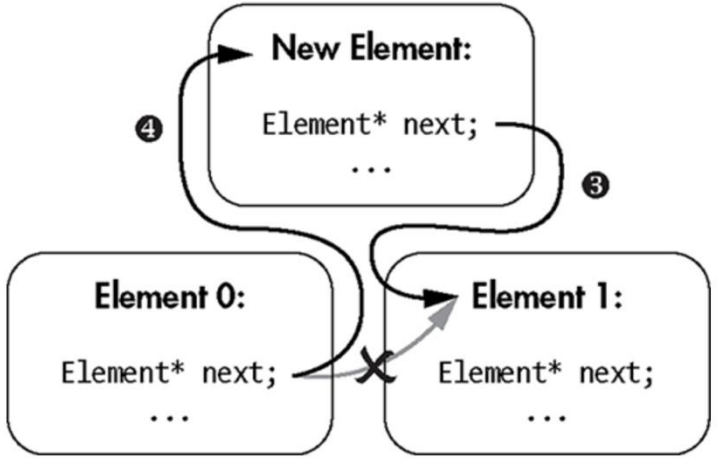

Figure 3-2: Inserting an element into a linked list

Employing References

- Pointers offer flexibility (but come with safety risks)
- References are safer when reseating or nullability isn't needed

ch3 > empRef.cpp > main()

```cpp
#include <cstdio>

int main(){
    int original = 100;
    int& original_ref = original;
    printf("Original:  %d\n", original);
    printf("Reference: %d\n", original_ref);

    int new_value = 200;
    original_ref = new_value;
    printf("Original: %d\n", original);
    printf("New Value: %d\n", new_value);
    printf("Reference: %d\n", original_ref);
}
```

ch3 — -zsh — 62×8

```
% ./empRef
Original:  100
Reference: 100
Original: 200
New Value: 200
Reference: 200
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch3
%
```

this Pointer

- this: a pointer to the **current object** inside a method
- implicit in most cases – explicitly used when disambiguation is needed
- When to use: to distinguish between member variables and method parameters

```
bool set_year(int year) {
  if (year < 2019) return false;
  this->year = year;  // Disambiguates from parameter
  return true;
}
```

const Correctness

- const means “I promise not to modify this.”
- Used to protect variables, parameters, or methods from accidental changes
- promotes safety and clarity: (context below)

```
1
2   //variable
3   const int max_value = 100;
4
5   //function parameters
6   void print(const std::string& message);
7
8   //Class method (promise not to modify the object
9   int get_year() const;
```

Member Initializer Lists

- Initialize class members before **constructor** body runs
- Required for **const** members and **recommended** for performance

```
class Example {
    public:
      Example(int v) : value{v} {}  // Member initializer list
    private:
      const int value;
    };
```

Type Deduction with auto

Auto

- auto lets compiler infer the type form context
- Reduces redundant type declarations

```
auto x = 42;  //int
auto name = "Ada"; //const char*
```

Benefits:

- Avoids verbose / complex type names
- Keeps code clean and maintainable
- Still strongly typed – type is just inferred, not dynamic

Auto and Reference Types

Auto can be combined with (for clarity and control):

```
auto year = 2019;                // int
auto& year_ref = year;           // int&
const auto& year_cref = year;    // const int&
auto* year_ptr = &year;          // int*
const auto* year_cptr = &year;   // const int*
```

auto and Code Refactoring

- Why use auto?
  - Simplifies code during refactoring (use consistently)
  - Adapts to changes in types automatically
  - Reduces the risk of introducing bugs when types evolve