

## Fundamentals and Machine Model

### Machine/Memory Model and the Function Call Stack

**Object:** a piece of data that's stored at a particular location in memory during runtime.

**Variable:** a *name* in source code that is associated with an object at compile time.

❗ Not all objects are associated with variables; e.g. dynamically-stored objects and string literals are not.

❗ The value stored by a variable's memory object may change, but the association between a variable and an object itself can only change when the variable goes out of **scope**.



Static objects "live" for essentially a program's runtime. Local objects' lifetimes are tied to scope (e.g. a block of code or pair of curly braces). Dynamic objects are manually created/destroyed.

❗ Objects declared in a loop body (between the {} ) are created/destroyed each time the loop repeats.

**Atomic (primitive) types:** objects that can't be subdivided into smaller objects; includes `int`, `double`, `bool`, `float`, `char`, and all pointer types. Atomic objects are default-initialized to undefined values.

The memory allocated to store a function's parameters and local variables during runtime is called a **stack frame** or activation record. The memory frame for the most-recently called function is added to the "top" of the **function call stack** and is destroyed when the function returns ("Last In First Out" ordering).

❗ Assignments inside of `return` statements (e.g. `return x = y;`) "take effect" before the return.

### Procedural Abstraction and Program Design

**Procedural Abstraction** involves using functions to break down a complex procedure into sub-tasks and separate the interface of a procedure (what it does) from implementation (how it works).

**Interface examples:** declarations in `.h` files, valid/invalid inputs, RME statements, *signature* (function name and parameter types), return type, and ADT representation invariants.

**Implementation examples:** definitions in `.cpp` files and code/comments inside function bodies.

## Pointers, Arrays and References

A **pointer** is a type of object that stores another object's memory address as its value.

❗ An `int*` pointer can *only* point to an `int`; an `int**` pointer can *only* point to an `int*`; and so on. (Trying to, for example, make an `int*` pointer point to a `double` will cause a compile error.)

**Dereferencing:** getting the object at an address. Note that the `*` operator is used both to declare pointers and to dereference them (and the `&` operator is used both to get an object's address and to declare references).

```
1 int x = 3; int y = 4;
2 int* ptr1 = &x; int* ptr2 = &y;
3 int** ptr1_ptr = &ptr1;
4 ptr2 = ptr1; // copies x's address from ptr1 to ptr2
5 ptr1 = &y; // ptr1 now points at y
6 **ptr1_ptr = 6; // now y == 6
7 *ptr2 = 2; // ptr2 still points to x, so now x == 2
```

❗ Printing a non-`char` pointer prints the address that it stores.

```
1 int x = 5; int& a = x;
2 int* y = &x; // creates pointer to x
3 int* z = y; // creates another pointer to x
4 int* ab = z; // creates reference b to pointer z
5 cout << *b << endl; // Prints 5
6 cout << y << endl; // prints 0x2710
7 cout << &(*z) << endl; // equiv. to cout << &x
8 cout << *(&z) << endl; // equiv. to cout << z
```

**Null pointer:** a pointer that holds address `0x0` (which no object can be located at) and implicitly converts to `false`. Any pointer can be nulled by setting it equal to `nullptr` (or `0`, or `NULL`).

### Differences Between Pointers and References:

- References are aliases for existing objects, while pointers are distinct objects with distinct values.
- Pointers must be dereferenced to access the objects they point at, while references are used "as-is".
- You can change what a (non-`const`) pointer points to, but you can't change what a reference refers to.

### Common Pointer/Reference Errors

⚠ Dereferencing an uninitialized pointer results in undefined behavior, as (like all atomic objects) pointers that aren't explicitly initialized are default-initialized to an undefined value (*not* `nullptr`).

⚠ Dereferencing a null pointer also leads to undefined behavior (almost always a program crash).

❗ An uninitialized reference or a reference-to-non-`const` that's bound to a "literal" value won't compile.

⚠ If a function returns a pointer or reference to one of its local variables (which die when the function returns), dereferencing that pointer or using that reference produces undefined behavior.

```
1 int* danglingPtr(int x) { return &x; } // BUGGY
1 int& danglingRef(int x) { return x; } // BUGGY
```

⚠ Be careful with mixing incrementing and dereferencing (and parentheses).

```
int x = 5;
int* ptr = &x;
// Output: 5 and 6
cout << (*ptr)++ << endl;
cout << x << endl;
// ptr == &x

int x = 5;
int* ptr = &x;
// Output: 5 and 5
cout << *ptr++ << endl;
cout << x << endl;
// ptr == junk (++&x)

int x = 5;
int* ptr = &x;
// Output: 6 and 6
cout << ++*ptr << endl;
cout << x << endl;
// ptr == &x

int x = 5;
int* ptr = &x;
// Output: 6 and 6
cout << ++(*ptr) << endl;
cout << x << endl;
// ptr == &x
```

### Arrays and Pointer Arithmetic

**Arrays:** fixed-size containers that store objects of the same type (and same size) in contiguous memory.

```
int A[3] = {1,2}; // {1,2,0}
int B[3] = {}; // {0,0,0}
int C[] = {1,2}; // size = 2

int D[2][3] = {1,2,3}; // {{1,2},{3,0}}
int E[1][3] = {1,2,3}; // {{1,2,3}}
int F[3]; // CAUTION: uninitialized!

int G[]; // ERROR: unclear size
int H[2][1] = {1,2,3,4}; // Same
int I[] = {}; // Same
```

**Array decay:** using an array in a context where a value is required causes the compiler to convert the array into a pointer to its first element. Array decay is why it's necessary to pass an array's size separately from the array to a function (or to indicate the end of an array with a *sentinel character* like C-strings do).

⚠ Dereferencing a pointer that goes past the bounds of an array results in undefined behavior. But merely *using* a pointer that goes just past the end of an array without dereferencing it is well-defined.

```
1 void reverseArray(int arr[], int size) {
2   for (int i = 0; i < (size / 2); ++i) {
3     int temp = arr[i];
4     arr[i] = arr[(size - i - 1)];
5     arr[(size - i - 1)] = temp;
6   } // Note: arr[i] == *(arr + i) == i[arr]
7   // Therefore, &arr[i] == (arr + i)
```

Passing an array by value passes a pointer to its first element by value, so functions with array parameters like this one actually have pointer parameters.

❗ The number of elements in an array `arr` is equal to `(sizeof(arr) / sizeof(*arr))`.

❗ `cout << &arr[0]` and `cout << &arr` would also print `0x1000`.

❗ You can create references to array elements.

```
1 int arr[3] = { 5, 10, 15 };
2 int* ptr = &arr[2];
3 int* ptr2 = (arr + 1);
4 cout << arr << endl; // prints 0x1000
5 cout << &ptr2 << endl; // prints 0x1014
6 cout << ptr[-1] << endl; // prints 10
```

```
1 // Mainly for pointers into the same array
2 int arr[4] = { 5, 10, 15, 20 };
3 int* ptr1 = &arr[0]; *ptr2 = &arr[3];
4 cout << *arr << endl; // prints 5
5 cout << (ptr2 - ptr1) << endl; // prints 3
6 cout << (ptr1 - ptr2) << endl; // prints -3
7 assert(ptr1 < ptr2); // true
8 ptr1 = 2; // ptr1 now points at arr[2]
```

```
1 int arr[4] = { 1, 2, 3, 4 };
2 int (*arr_ptr)[4] = &arr; // pointer to entire array
3 cout << (*arr_ptr)[2] << endl; // prints 3
4 // ++arr_ptr would increment by the size of 4 ints
```

**Pointer arithmetic:** adding an integer `n` to a pointer yields a pointer that is `n` objects forward in memory.

**Pointer subtraction:** Subtracting two pointers of the same type yields an integer (possibly a negative one) equal to the *number of objects* between them.

**Pointer comparison:** comparing pointers of the same type compares the addresses they store.

Using `&` on an array (without an index) creates a pointer to the entire array, not a pointer to the first element or a pointer to a pointer.

**Array traversal:** arrays can be traversed by index or pointer (differentiated by what gets incremented).

```
Traversal By Pointer: Pattern 1
1 int computeRange(const int arr[], int N) {
2   const int min = arr; // Need const here
3   const int max = arr;
4   const int end = (arr + N);
5   // end is actually "1-past-the-end"
6   for (const int* p = arr; p < end; ++p) {
7     if (*p < min) { min = p; }
8     if (*p > max) { max = p; }
9   } // "walk" the pointer across arr
10  return (*max - *min);
11 }
```

```
Traversal by Pointer: Pattern 2 (C-String Sanitization)
1 void sanitize(char username[], char to_remove) {
2   char* slow = username, *fast = username;
3   while (*slow && *fast) { // while not '\0'
4     if (*fast != to_remove) {
5       *slow = *fast;
6       ++slow; // ++slow only if we copy
7     }
8     ++fast; // ++fast every loop
9   }
10  *slow = '\0'; // null-terminate when done
11 } // NOTE: '\0' is the only char considered "false"
```

## The const Keyword

The `const` type qualifier stops objects from being modified after initialization. Note: `const` scalars must be explicitly-initialized to compile, and `const` class-type objects must have their data members initialized.

**const pointers:** pointers that can modify what they point at but cannot be re-pointed to different objects.

**Pointer-to-const:** read-only pointers; pointers that can be re-bound but can't modify what they point at.

❗ A `const` pointer must be initialized to compile, but a pointer-to-`const` doesn't need to be.

**Reference-to-const:** a read-only alias.

**const array:** an array of `const` elements. Note that the placement of `const` matters for arrays of pointers.

```
const int* A[] = { ... }; // pointer-to-const array
int* const B[] = { ... }; // const pointer array
```

### const Conversions and Passing

The compiler treats every pointer-to-`const` as if they point to a `const` object and every reference-to-`const` as if they're aliased to a `const` object. It won't allow conversions that could bypass existing `const` protections (so, e.g., you can assign a `const` pointer to a pointer-to-`const`, but the converse is *not* true).

```
1 void foo(string& a) { ... }
2 void bar(string b) { ... }
3 void func(const string& c) { ... }
4 const string s = "Hello World";
5 bar(s); func(s); // both OK
6 foo(s); foo("Hello"); // ERRORS

1 const int x = 3;
2 int y = x; // OK
3 const int* cptr = &x; // OK
4 const int& cref = x; // OK
5 int* ptr = cptr; // ERROR 1
6 int& ref = cref; // ERROR 2

1 int x = 2, y = 5;
2 const int *x_ptr = &x;
3 int *y_ptr = &y;
4 *y_ptr = *x_ptr; // OK
5 *y_ptr = x_ptr; // ERROR (even though x isn't const!) *
```

- Pass by pointer/reference:** if you need to modify the original object (as opposed to a local copy).
- Pass by value:** if an object is small (e.g., an `int`) and you can't/don't need to modify the original.
- Pass by pointer/reference-to-const:** if you want to pass a large object without modifying it.

## Strings, Streams and I/O

### Creating/Using C-Strings and Strings

```
1 char color[] = "0027UC"; // Create 7-element array (including \0) and copy a string literal to it
2 const char* cstr = "abcd"; // Only works for string literals; use .c_str() on string variables
3 cout << cstr << " " << *cstr << " " << &cstr[0] << endl; // prints "abcd aabcd"
4 cout << (cstr + 1) << " " << *cstr + 1 << " " << *cstr + 1; // prints "bcd b 98" ('a' == 97)
5 string xyz = string(cstr); // Explicitly copy cstring to a string (implicit copy would work too)
```

	Length	Copy Value	Index	Concatenate	Compare
<code>&lt;string&gt;</code>	<code>str.length()</code>	<code>str1 = str2;</code>	<code>str[i];</code>	<code>str1 += str2;</code>	<code>str1 != str2;</code>
<code>&lt;cstring&gt;</code>	<code>strlen(cstr);</code>	<code>strcpy(cstr1, cstr2);</code>	<code>cstr[i];</code>	<code>strcat(cstr1, cstr2);</code>	<code>strcmp(cstr1, cstr2);</code>

### Streams and File I/O

stdin Redirection	stdout Redirection	Pipeline	Combined Redirection
<code>./main.exe &lt; input.txt</code>	<code>./main.exe &gt; output.txt</code>	<code>./output.exe   input.exe</code>	<code>./main.exe &lt; input.in &gt; output.out</code>

```
File I/O Ex 1: Print Lines From File
1 #include <fstream> // defines if/ofstream
2 int main() {
3   ifstream inFS; // or inFS("file.txt");
4   inFS.open("file.txt");
5   if (!inFS.is_open()) { return 1; }
6   string str; // defaults to empty string
7   while (getline(inFS, str)) {
8     cout << str << endl;
9   } // could close inFS via inFS.close();
10  // inFS also closes when scope ends

Ex 2: Copy One File's Contents to Another
1 #include <fstream> // defines stringstream
2 void copyFile(string file_in, string file_out) {
3   // file streams also accept C-strings as arguments
4   ifstream inFS(file_in);
5   ofstream outFS(file_out);
6   string input_str;
7   while (inFS >> input_str) {
8     outFS << input_str << endl;
9   } // could use '\n' instead of endl
```

❗ The insertion `<<` and extraction `>>` operators "stop" at any white space (spaces, line breaks, etc).

**istreamstream:** an object that "simulates" *input* with a string as its source. Note: an `istreamstream`, an `ifstream` and `cin` can all be passed to a function with a `std::istream&` parameter.

**ostreamstream:** an object that captures *output* and stores it in a string. Note: an `ostreamstream`, an `ofstream` and `cout` can all be passed to a function with a `std::ostream&` parameter.

### Command-Line Arguments

**argc:** an int parameter of `main()` representing the number of a command's arguments.

**argv:** an array of the arguments passed to a program. (Technically, `argv` is an array of pointers to C-strings—so `argv` is passed to `main()` as a pointer to an array of pointers to C-strings).

```
1 #include <string> // defines stoi()/stod()
2 int main(int argc, char* argv[]) { // char** argv also OK
3   if (string(argv[1]) == "add") {
4     int sum = 0;
5     for (int i = 2; i < argc; ++i) {
6       sum += stoi(argv[i]);
7     }
8     cout << "Sum: " << sum << ", argc: " << argc << endl;
9   } // pay attention to where the "actual" arguments start
10  // Also remember to use stoi()/string() when needed
```

## ADTs, Structs and Classes

### C-Style Structs and ADTs

A **struct** is a class-type object composed of member subobjects (heterogeneous data). They're passed by value by default, and they support assignment and initialization via the `=` operator. A `struct` or `class` object can also be declared as `const`, which prevents it and all of its data members from being modified.

❗ You cannot call non-`const` member functions on a `const` instance of a `class` or `struct`. Also, you can't call non-`const` member functions from within a `const` member function.

**Arrow `>>` operator:** shorthand for a dereference followed by member access. (`*ptr.x` == `ptr->x`;

❗ Without parentheses, the dot and arrow operators have greater precedence than dereferencing.

**Abstract Data Type:** a data type that separates its behavior and implementation. ADTs encompass both data and behaviors/functions that act upon it. Not all `struct`'s are ADTs, some are "plain old data".

### C++ Classes

In C++, the only real difference between classes and structs are that classes have `private` member access and `private` inheritance by default while `struct`'s default to `public` access/inheritance.

