C Basics 1 Extra: C vs. C++

1. **C vs. C++**
   - **history of C++**
   - **C standards++**
   - **procedural vs. OOP**
   - **C vs. C++ characteristics**
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O

C Basics 1 Extra: C vs. C++

8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Miscellaneous

Overview of C++

C++ was created by Bjarne Stroustrup, around 1985, to replace earlier C extensions he had created.

Since these earlier languages added **OOP** capabilities to C, they were often referred to as "C with Classes."

While C programs were originally nearly runnable in C++, the two languages diverged quite a bit in the late 1990's.

In particular, C++ has been extended very significantly since it was created, with modern C++ being vastly larger and more complex than it was originally (and than C).

Nonetheless, it remains relatively backwards compatible with C.

Overview of C++ (contd.)

The C++11 standard states:
"C++ is a general purpose programming language based on the C programming language as specified in ISO/IEC 9899:1999 (hereinafter referred to as the C standard). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities."

Overview of C++ (contd.)

Wikipedia says this about C-C++ compatibility:
"Most C code can easily be made to compile correctly in C++ but there are a few differences that cause some valid C code to be invalid or behave differently in C++. For example, C allows implicit conversion from void* to other pointer types but C++ does not (for type safety reasons). Also, C++ defines many new keywords, such as new and class, which may be used as identifiers...in C.... Some incompatibilities have been removed by the 1999 revision of the C standard (C99).... On the other hand, C99 introduced a number of new features that C++ did not support that were incompatible or redundant in C++.... Some of the C99-introduced features were included in...C++11.... However, the C++11 standard introduces new incompatibilities, such as disallowing assignment of a string literal to a character pointer, which remains valid C."

Overview of C++ (contd.)

C++ is a very popular and much used programming language, always near the top of programming language usage surveys.

It finds much of its use in GUI/graphics applications, particularly where performance is critical (making Java unacceptable).

Despite its heavy real-world use, C++ has received a great deal of criticism, with many people calling it a bad language.

The most common complaints center on its size and complexity, since the number of **non-orthogonal** features in C++ means that in practice programmers use only limited subsets of the language, reducing **readability**.

Having multiple totally different ways to accomplish something basic is not a clear advantage in a programming language.

C++ Language Standards

1985: Stroustop publishes *The C++ Programming Language*.

C++98: C++ was first standardized, by ISO, in 1998.

Problems with C++98 were fixed with a revised standard in 2003.

C++11: released in 2011, with many additions to the core language and the standard library.

C++14: minimal extensions were made in 2014.

C++17: the latest revision was completed in 2017.

Procedural vs. OOP

C++ was originally developed as an extension of C, so it shares much syntax with C, such as declarations, control constructs, operators, etc.

C is a purely **procedural language**, however, while C++ supports both *procedural* and **object-oriented programming** (OOP) styles.

In a procedural language like C, programs are structured in terms of **functions** (**subroutines**) operating on **data** held in **variables**.

The focus is on the *actions* that need to be taken to accomplish the goals of the program.

With OOP, programs are structured in terms of classes of objects and the operations that can be done to them.

Procedural vs. OOP (contd.)

Procedural *functions* correspond to OOP *methods* (also known as member functions).

Procedural *variables* correspond to OOP *class instances* and/or *members* of instances.

Whether procedural or OOP approaches are best for a program is largely a matter of which makes it more natural to conceptualize program structure.

If the focus is on complex actions (algorithms), then procedural may be better.

If the focus is on manipulating different data objects, then OOP may be better.

Procedural vs. OOP (contd.)

Often it is more an issue of how program **modularity** can best be accomplished.

Pure OOP requires that every function be a method inside a class, which can lead to non-intuitive design.

E.g., the `Math` class in Java, which is a class that exists simply to hold Java's basic mathematical functionality like `min()`, etc.

`Math` makes no sense as a *class*, but Java must have it because every function has to be a method inside a class in Java.

Likewise, while OOP encapsulation can be very useful, real-world OOP languages typically have various way to get around strict encapsulation, since it can make OOP programs impractical (e.g., **public** members and **friend functions** in C++).

Procedural vs. OOP (contd.)

When moving from an OOP language to a procedural one, it is important to understand that OOP *encapsulation* is not the only way to implement *modularity* in programs.

In C, modularity is largely based on decomposition into *files*.

E.g., a "**global variable**" in a file is actually global only to the functions defined in that *same file*.

While its **scope** can be extended to functions in other system files by using the `extern` keyword, this can be disallowed by using the keyword `static` with the global definition.

Function scope (visibility) can be similarly controlled.

The analogy to *public* and *private* OOP members should be clear.

C vs. C++

	C	C++
Programming Paradigm	pure procedural	procedural or OOP
Composite Data Types	structs	structs and classes
Type Inheritance	no	**derived class** from *multiple* **base classes**
Encapsulation Mechanism	files	files and classes
Compilation Modularity	files	files
Memory Management (heap)	manual `malloc()`, `realloc()`, `free()`, etc.	manual (mostly) C functions available, but prefer: `new`, `delete` (required with classes)

C vs. C++ (contd.)

	C	C++

Pointers	explicit, manipulable:
	defined: *basetype*`*`
	`int *ip, i;`
	`ip = &i + 1;`
		"raw" pointers same,
		C++11 added:
		safe pointers
		(prevent **memory leaks**)
		Also, **reference** types,
		implicit pointers,
		not manipulable

Invalid Pointer	`NULL` `(void *)0`	`NULL` (macro) and keyword `nullptr`
Numeric Types	integer: `char`, `short`, `int`, `long`, `long long` real: `float`, `double`, `long double` (C99) other: `_Bool` (C99) sign: `signed`, `unsigned` minimum sizes only	mostly same, Boolean: `bool`

C vs. C++(contd.)

	C	C++
Boolean Type	<C99 no, use `int`'s C99: `_Bool` (0/1 `int`) 0 is false 1 is standard true, but any $\neq 0$ considered true C99: `stdbool.h`: `bool`, `false`, `true`	type `bool` values `false`, `true` (effectively 0 and 1)
Characters	ASCII encoding single byte integer constants: `'a'`, `'\141'` C99: wide char's C11: Unicode	similar: byte `char` types, wide chars's, UTF-8, 16, 32 char types

C vs. C++(contd.)

	C	C++
Strings	non-builtin type: type `char*` (`char` array, requires '\0' sentinel) constant/literal: `"abc"`	possibly multiple types: <C++98: C style only, ≥C++98: `std::string`, used in C++ library
String Comparisons	`string` library functions e.g., `strcmp()`	C lib for C strings C++ lib for `std::string` e.g., `std::string::compare()` (overloaded)

C vs. C++ (contd.)

	C	C++

Arrays	access: `arr[i]`
	declaration:
	`int iarr[`*constant*`]`
	static size
	no bounds checking
	C99: **variable-length** arrays:
	`int iarr[`*expression*`]`
	C11: VLA made *optional*
		basic arrays same,
		also `std::array`
		
		VLA not standard,
		GCC extension allows

Dynamic Arrays (heap)	Use malloc(), etc:
	`int *iarr =`
	`malloc(n*sizeof(int))`
		can do same as C
		
		Also: `new[]`, `delete[]`
		and library **containers**
		like `std::vector`

C vs. C++ (contd.)

	C	C++
Implicit Type Conversions (**coercion**)	extensive: *promotion & demotion* may not preserve values: `unsigned int`→`signed int`	same
Explicit Type Conversions (**casting**)	yes: (*type*) *var/expr*	C-style plus: `static_cast`, `dynamic_cast`, etc.

C vs. C++ (contd.)

	C	C++
Main (entry point)	`int main(int argc, char *argv[])` `int main()`	same
Operators	assignment: `=`, `+=`, etc. arithmetic: `+`,`++`, etc. relational: `==`, `!=`, etc. logical/bitwise: `&&`, `&`, etc.	same same same same
Selection Constructs	`if`, `else`, `switch`	same
Looping Constructs	`for`, `while`, `do-while`	same
Global Variables	yes ("**external**" definitions)	same, but: not used with OOP
Automatic Variable Initialization	generally no: global and `static` only	same, except: class instances invoke constructor

C vs. C++ (contd.)

	C	C++
Subroutines	functions	functions class methods
Subroutine Values	any type plus `void`	same
Parameter Passing	call-by-value: but pointer params act like call-by-ref	call-by-value: but reference params act like call-by-ref
Generic Functions	no, must use `void*` pointers C11: generic selector `_Generic`	yes: templates
Function Overloading	no	yes
Operator Overloading	no	yes

C vs. C++ (contd.)

	C	C++
Function Arguments (Callbacks)	yes: `hofunc(void (*h)(int));`	same
Variadic Functions	yes: (`stdarg.h`) `printf(char *fmt, ...);`	similar
Optional or Default Parameters	no	yes
Preprocessor	yes	yes
Header Files	yes `.h` given explicitly	yes `.h` not written C library headers as: `string.h` → `cstring`

C vs. C++ (contd.)

	C	C++
Namespaces	not really, since fixed: "ordinary identifers", **tags** for structs/unions/enums, members, **labels**	yes
Exception Mechanism	no	yes: `try`, `catch`, `throw`
Input/ Output	library functions from `stdio`: `printf()`, `scanf()`, etc. (many functions) e.g., `printf("Val:%f\n",x);` `FILE *inf = fopen("test","r");`	streams from `iostream`: `cin`, `cout`, `cerr` e.g., `cout<<"Val:"<<x<<endl;` from `fstream`: `ifstream inf; inf.open("test");`

C vs. C++ (contd.)

	C	C++
Language Libraries	**C Standard Library** required by standards referred to as `libc` modest: IO, strings, math, etc.	**C++ Standard Library** required by standards referred to as `libstdc++` large: many classes, generics and containers
Concurrency: via Multiple Processes	via **system calls** (`fork()`, etc.)	same
Concurrency: via Multiple OS Threads	via **system calls** (`pthread_create()`, etc.) **C11** added: `threads.h` with: `thrd_create()`, `thrd_join()`, etc.	yes: via syscalls plus `std::thread` and associated classes: with constructors and methods like `join`, `detach`, etc.