Ch 2: Types

CSCI 330

Overview

- Fundamental Types (int, float, char, etc.)
- User-Defined Types (struct, class, enum)
- Control Flow Structures (loops, branches)

What is a Type:

- A **type** defines how an **object** is interpreted and used by the **compiler**
- **Every object** in a C++ program has an associated type

Fundamental (primitive / built-in) Types

- Core language types: int, char, bool, float
- Always available; essential for performance and portability

Low-Level Control

- Hardware mapping directly to machine-level types
- Enables fast, memory-efficient operations
- Gives explicit control over data layout and behavior

High-Level Portability

- Abstracted just enough for **cross-platform consistency**
- Code runs reliably across **different OS and hardware**
- Supports "write once, run anywhere" development

Integer Types

- Store **whole numbers** (no decimal point)
- Support both **positive and negative values** (signed) or **only non-negative** (unsigned)

Integer Types, Sizes, and Format Specifiers

Type	Signed	32-bit OS Size	64-bit OS Size	printf Format
short	Yes	2 bytes	2 bytes	%hd
unsigned short	No	2 bytes	2 bytes	%hu
int	Yes	4 bytes	4 bytes	%d
unsigned int	No	4 bytes	4 bytes	%u
long	Yes	4 bytes	4 bytes Win | 8 bytes Linux/MacOS	%ld
Unsigned long	No	4 bytes	4 bytes Win | 8 bytes Linux/MacOS	%lu
Long long	Yes	8 bytes	8 bytes	%lld
Unsigned long long	No	8 bytes	8 bytes	%llu

Literals

- A literal is a hardcoded value in your code.
- C++ integer literals:

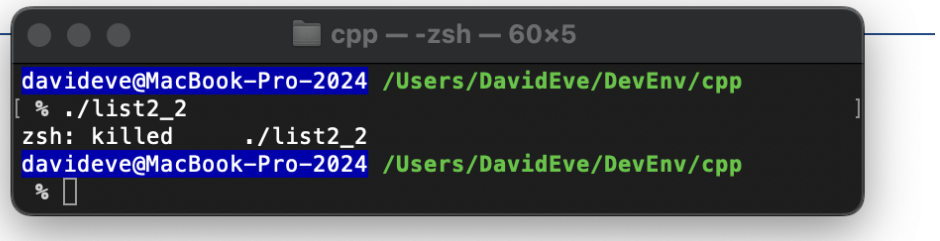
Format	Prefix	Example
Binary	0b	0b1010
Octal	0	012
Decimal		10
Hex	0x	0xa

```cpp
#include <cstdio>

int main() {
    unsigned short a = 0b10101010;
    printf("%hu\n", a);
    int b = 0123;
    printf("%d\n", b);
    unsigned long long d = 0xFFFFFFFFFFFFFF;
    printf("%llu\n", d);
}
```

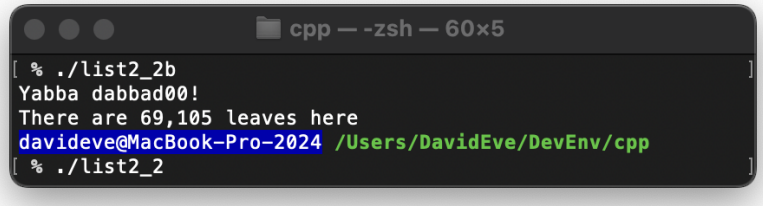
```
davideve@MacBook-Pro-2024 /Users/DavidEve/DevEnv/cpp
 % ./list2_1
170
83
72057594037927935
```

What? It didn’t work?

```cpp
#include <cstdio>

int main() {
    unsigned int a = 3669732608;
    printf("Yabba %x!\n", a);
    unsigned int b = 69;
    printf("There are %u,%o leaves here\n", b, b);
    return 0;
}
```




```cpp
#include <cstdio>
#include <cinttypes>  // for PRIu32

int main() {
    uint32_t a = 3669732608U;
    printf("Yabba %" PRIx32 "!\n", a);

    uint32_t b = 69;
    printf("There are %" PRIu32 ",%o leaves here\n", b, b);
    return 0;
}
```



Integer Suffixes

- C++, by default, uses the smallest type that can hold the value
- Suffix can be used to force a different (larger) type.

  Example: the literal 112114 is treated as an int by default

Literal	Resulting Type
112114	int (default)
112114U or 112114u	int
112114L	long
112114LL	long long
11211114LLU	unsigned int

Floating-Point Types

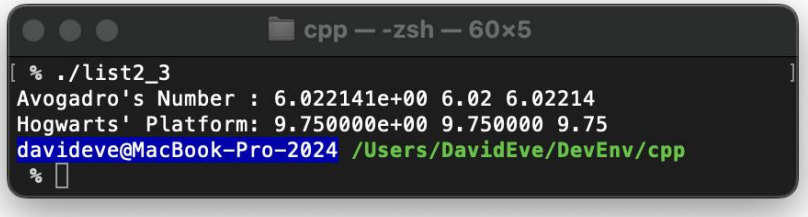
- Used to store approximations of real numbers (e.g., 0.33333, 98.6, π)
- Exact representation impossible due to finite memory
- Floating-point math may produce unexpected rounding errors

Type	Description	Typical Size
float	single precision	4 bytes
double	double precision	8 bytes
long double	extended precision	8+ bytes (depending on system and compiler)

Floating-Point Format Specifiers

Specifier	Description	Example
%f	Fixed-point notation (default 6 digits)	3.141593
%.2f	Fixed-point, 2 decimal places	3.14
%e	Scientific notation	3.141593e+00
%g	Shorter of %f or %e, trims trailing zeros	3.14, 3.1e+05
%a	Hexadecimal floating-point	0x1.91eb86p+1

```
1   #include <cstdio>
2
3   int main(){
4       double an = 6.0221409;
5       printf("Avogadro's Number : %e %.2f %g\n", an, an, an);
6       float hp = 9.75;
7       printf("Hogwarts' Platform: %e %f %g\n", hp, hp, hp);
8   }
9
```



Character Types

- C++ supports 6 character types for storing text and symbols:

Type	Description	Size
char	Default (1byte), may be signed/unsigned	Narrow
signed char	Guaranteed signed 1-byte	Narrow
unsigned char	Guaranteed unsigned 1-byte	Narrow
char16_t	2-byte Unicode (UTF-16)	Wide
char32_t	4-byte Unicode (UTF-32)	Wide
wchar_t	Local-dependent wide character	Wide

Character Literals

- All character literals use **single quotes** (' '), not double quotes.
- Use prefixes for **non-ASCII or wide character encodings**.
- Character literals are **distinct from string literals** ("J").

```
char initial = 'J';
```

Literal	Type	Description	Use
‘J’	char	Default, 1 byte	ASCII and Simple text
L’J’	wchar_t	Wide Character (local-based)	Wide characters (platform/local specific)
u’J’	char16_t	UTF-16, 2 bytes	UTF-16 encoding (Cross-platform Unicode)
U’J’	char32_t	UTF-32, 4 bytes	UTF-32 encoding (1:1 Unicode codepoints)

Escape Sequences

Value	Escape Sequence
Newline	\n
Tab (horizontal)	\t
Tab (vertical)	\v
Backspace	\b
Carriage return	\r
Form feed	\f

Value	Escape Sequence
Alert	\a
Backslash	\\
Question mark	\?
Single quote	\’
Double quote	\”
The null character	\0

Format Specifiers

- Format specifiers are not cosmetic – they’re vital for correctness, clarity, and compatibility!!
- Always match the specifier to the variable type:
  - %c used for char
  - %lc used for wchar_t

Boolean Types

- Boolean literal: true, false
- No specific format specifiers for Boolean
- int format specifier %d within printf returns 1 for true, 0 for false

```
1   #include <cstdio>
2
3   int main() {
4       bool b1 = true;   // b1 is true
5       bool b2 = false;  // b2 is false
6       printf("%d %d\n", b1, b2);
7   }
8
9
```

```
cpp — -zsh — 56×5
davideve@MacBook-Pro-2024 ~/DevEnv/cpp
 % ./boolExample
1 0
davideve@MacBook-Pro-2024 ~/DevEnv/cpp
 %
```

Comparison Operators

- Operators are **functions** that act on **operands** (i.e., values or variables).
- In this context, the **comparison operators** return a **bool (T/F)**

```cpp
#include <cstdio>

int main() {
    printf(" 7 ==  7: %d\n", 7  ==  7);
    printf(" 7 !=  7: %d\n", 7  !=  7);
    printf("10 > 20: %d\n", 10 >  20);
    printf("10 >= 20: %d\n", 10 >= 20);
    printf("10 < 20: %d\n", 10 <  20);
    printf("20 <= 20: %d\n", 20 <= 20);
}

```

Boolean Comparison Operators

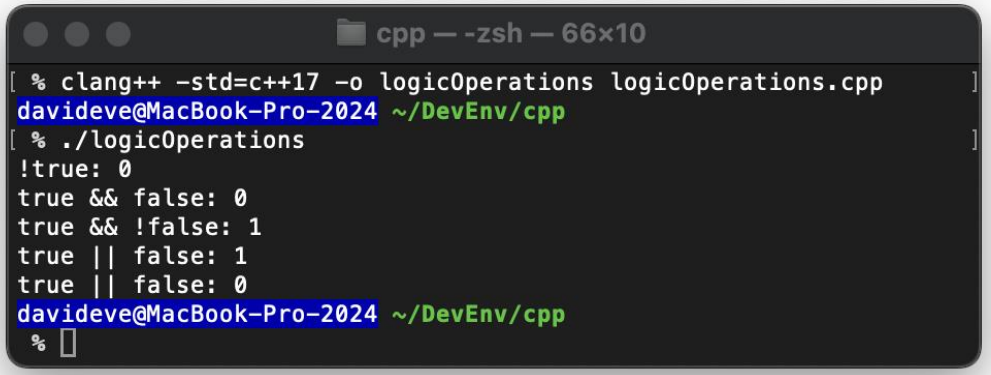
Operator	Meaning	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b
<=	Less than or equal	a <= b

Logical Operators

- Logical operators evaluate Boolean conditions (T/F)

Operator	Name	Arity	Returns	Example
!	Logical NOT	Unary	The operand is false	!true returns false
&&	Logical AND	Binary	Both operands are true	true && true
||	Logical OR	Binary	At least one operand is true	true || true

```cpp
#include <cstdio>

int main() {
    bool t = true;
    bool f = false;
    printf("!true: %d\n", t && f);
    printf("true && false: %d\n", t && f);
    printf("true && !false: %d\n", t && !f);
    printf("true || false: %d\n", t || f);
    printf("true || false: %d\n", f || f);
  }

```



std::byte Type

- Used to work directly with raw memory (buffers, hardware memory maps, file data)
- std::byte (from <cstddef> is a type-safe to handle raw memory (doesn't support arithmetic, beyond bitwise logical operations)
- Acts like a typeless collection of bits (ideal when data structure is unknown or undefined)

Type	Allows Arithmetic?	Safer
char	yes	no
unsigned char	yes	no
std::type	no	yes

size_t Type and sizeof: Measuring Memory

What It Is:

size_t (from <cstddef>) is the standard **unsigned type** used to represent:

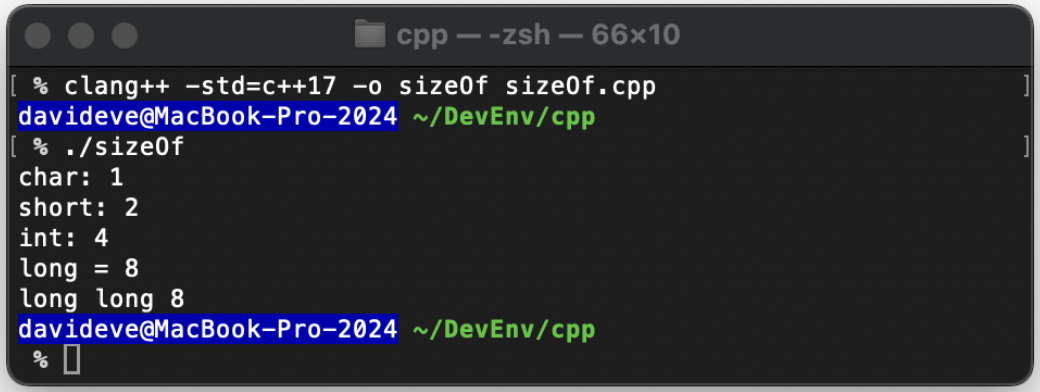
- The **size of objects in memory** (indexing arrays, buffers, containers)
- The **result of sizeof** (Unary operator that returns the number of bytes used by type or object)

Why Use size_t?

- Guaranteed to be large enough to hold the **maximum size of any object** on the system.
- On 64-bit systems, it's usually equivalent to unsigned long long.
- **Portable, implementation-dependent**: adapts to system architecture.

sizeof Format Specifiers (%zu)

```cpp
#include <cstddef>
#include <cstdio>

int main() {
    size_t size_c = sizeof(char);
    printf("char: %zu\n", size_c);
    size_t size_s = sizeof(short);
    printf("short: %zu\n" , size_s);
    size_t size_i = sizeof(int);
    printf("int: %zu\n", size_i);
    size_t size_l = sizeof(long);
    printf("long = %zu\n", size_l);
    size_t size_ll = sizeof(long long);
    printf("long long %zu\n", size_ll);
}
```

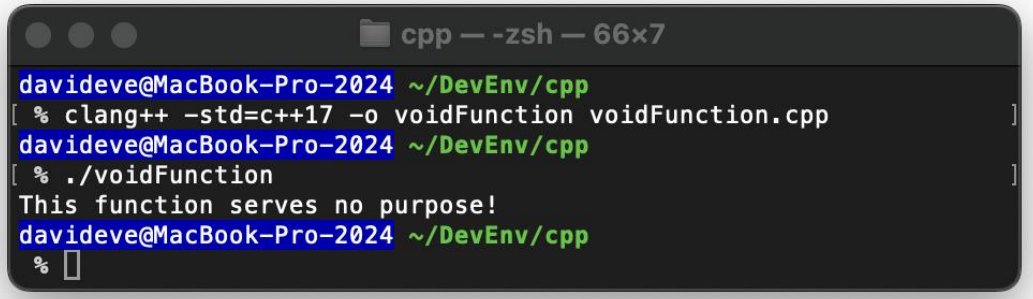


void

A special type with no values

Used in functions that don’t return anything

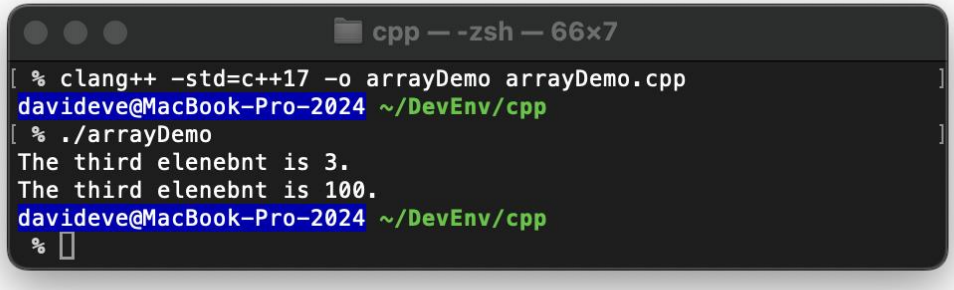
```cpp
#include <iostream>

void greetUser() {
    std::cout <<"This function serves no purpose!\n";
}

int main() {
    greetUser();
    return 0;
}
```



Arrays

- A sequence of elements of the same type stored in contiguous memory
- Arrays are zero-indexed (begin at 0 for first element)
- Array type = element type and number of elements
- Arrays are fixed size and typed at compile time

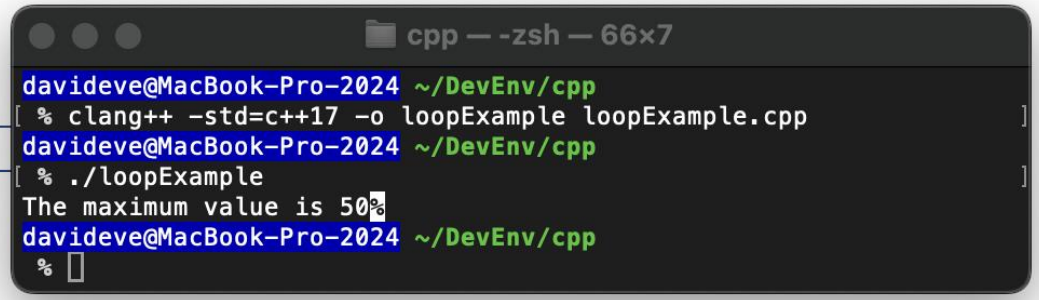
```cpp
#include <cstdio>

int main() {
    int arr[] = {1,2,3,4};
    printf("The third elenebnt is %d.\n", arr[2]);

    arr[2]= 100;
    printf("The third elenebnt is %d.\n", arr[2]);


}
```



For Loops

- Repeats a statement or block a specified number of times.
- Single statement contains: initialization, condition checking, and iterations

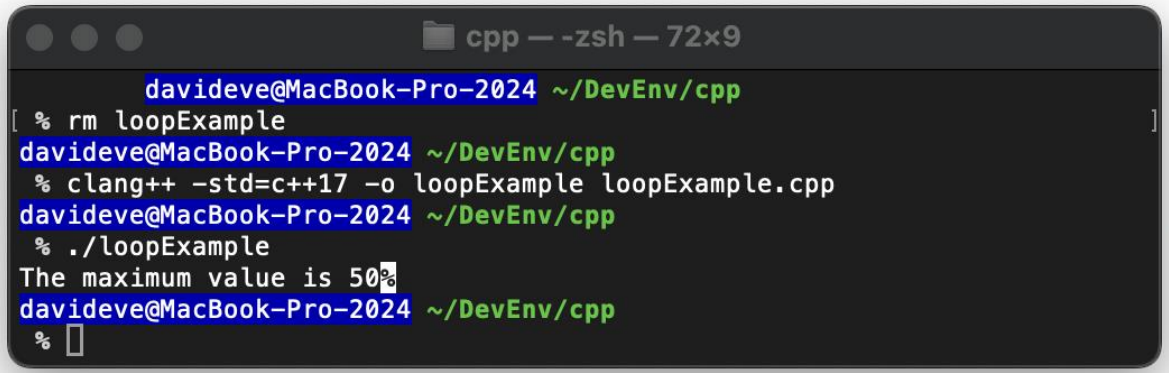
```
1   #include <cstdio>
2   #include <cstddef>
3
4   int main() {
5       unsigned long maximum = 0;
6       unsigned long values[] = {10, 50, 40, 22, 0};
7       for (size_t i=0; i < 5; i++){
8        if (values[i] > maximum) maximum = values[i];
9       }
10      printf("The maximum value is %lu", maximum);
11  }
12
13
```



Range-based For Loops

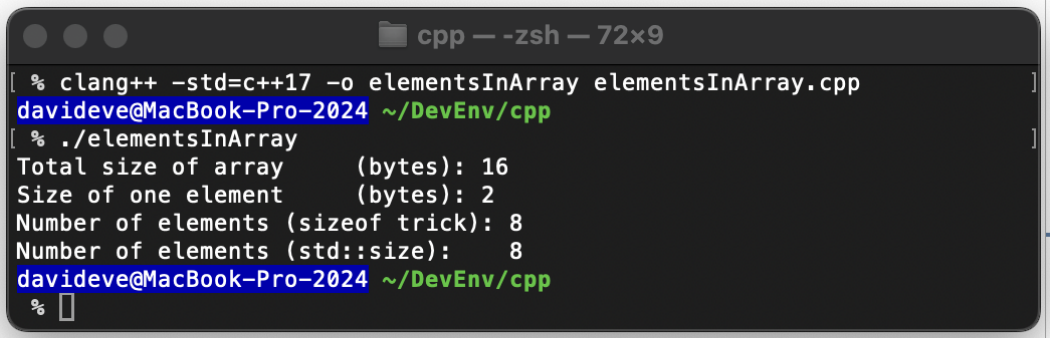
- Repeats a statement or block a specified number of times.
- Single statement iterates over values – much cleaner!

```cpp
#include <cstdio>
#include <cstddef>

int main() {
    unsigned long maximum = 0;
    unsigned long values[] = {10, 50, 40, 22, 0};
    for (unsigned long value: values){
     if (value > maximum) maximum = value;
    }
    printf("The maximum value is %lu", maximum);
}

```



Number of Elements in an Array

```cpp
#include <iostream>
#include <iterator> // For std::size (C++17 and later)

int main() {
    // Define a raw array of short integers
    short array[] = { 104, 105, 32, 98, 105, 108, 108, 0 };

    // Total size in bytes of the entire array
    size_t total_bytes = sizeof(array);

    // Size in bytes of one element (short)
    size_t element_size = sizeof(short);

    // Number of elements using sizeof trick
    size_t num_elements = total_bytes / element_size;

    // Number of elements using std::size (C++17+)
    size_t num_elements_std = std::size(array);

    // Print results
    std::cout << "Total size of array     (bytes): " << total_bytes << std::endl;
    std::cout << "Size of one element     (bytes): " << element_size << std::endl;
    std::cout << "Number of elements (sizeof trick): " << num_elements << std::endl;
    std::cout << "Number of elements (std::size):    " << num_elements_std << std::endl;

    return 0;
}
```



Determining Array Size in C++

Why sizeof(arr) / sizeof(arr[0]) Only Works with Raw Arrays

- **Raw arrays** have their full size known at **compile time**
  → sizeof(arr) returns total bytes, e.g. int arr[5] → 20 bytes (if int is 4 bytes)

- **Pointers** (e.g., int* ptr = new int[5]) only store an address
  sizeof(ptr) gives pointer size (e.g., 8 bytes), **not** array size

- Arrays **decay to pointers** when passed to functions
  sizeof(arr) inside a function becomes sizeof(pointer), not array size

- **Use std::size(arr)** (C++17+) as a safer, modern alternative for raw arrays

C-Style Strings (Null-Terminated Strings)

- A **C-style string** is a contiguous array of characters ending with a null byte (‘\0’)
- The null terminator marks the end of the string, not its size.
- C-style strings are stored in character arrays

```
char msg[] = "Hello";  // Stored as: {'H', 'e', 'l', 'l', 'o', '\0’}
```

- Because arrays are contiguous in memory, string functions (strlen, strcmp) can walk through each character until ‘\0’

String Literals in C++

- String Literals are sequence of characters in double quotes:

  char English[] = “a book holds a house of gold.”;

- They are null-terminated C-style strings stored in arrays
- To support Unicode, use prefix:
  - u”” UTF-16(char16_t)
  - U”” UTF-32 (char32_t)
  - L”” Wide characters (wchar_t)

Using Unicode Code Points

UnicodePoints.cpp > ...

```cpp
#include <iostream>
#include <iomanip>
#include <codecvt>
#include <locale>

int main() {
    const char* english = "A book holds a house of gold.";
    char16_t chinese[] = u"\u4e66\u4e2d\u81ea\u6709\u9ec4\u91d1\u5c4b"; // 书中自有黄金屋

    std::u16string u16str(chinese);
    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> cvt;
    std::string utf8 = cvt.to_bytes(u16str);

    std::cout << "English: " << english << "\n"
              << "Chinese: " << utf8 << "\n\n"
              << "Char  | Unicode\n----------------\n";

    for (char16_t ch : u16str) {
        if (ch == u'\0') break;
        std::cout << cvt.to_bytes(ch)
                  << "     | U+"
                  << std::hex << std::uppercase << std::setw(4)
                  << std::setfill('0') << static_cast<int>(ch) << "\n";
    }
}
```

```
cpp — -zsh — 62×18
davideve@MacBook-Pro-2024 ~/DevEnv/cpp
 % clang++ -std=c++17 -o unicodePoints unicodePoints.cpp
davideve@MacBook-Pro-2024 ~/DevEnv/cpp
 % ./unicodePoints
English: A book holds a house of gold.
Chinese: 书中自有黄金屋

Char  | Unicode
----------------
书     | U+4E66
中     | U+4E2D
自     | U+81EA
有     | U+6709
黄     | U+9EC4
金     | U+91D1
屋     | U+5C4B
davideve@MacBook-Pro-2024 ~/DevEnv/cpp
 %
```

Format Specifier

- use %s in printf to format C-style strings (char*)

```
char house[] = “a house of gold.”;
printf(“A book holds %s\n”, housle);

Output:  A book holds a house of gold.
```

- Multi-Line String Literals: Adjacent string literals automatically concatenated (useful for splitting long strings/readability)

```
char house[] = “A ”
“house “ “of”
“gold.”;

Output: A house of gold.
```

- Printing Unicode (**char16_t**, **wchar_t**) needs **wprintf** from **<cwchar>**

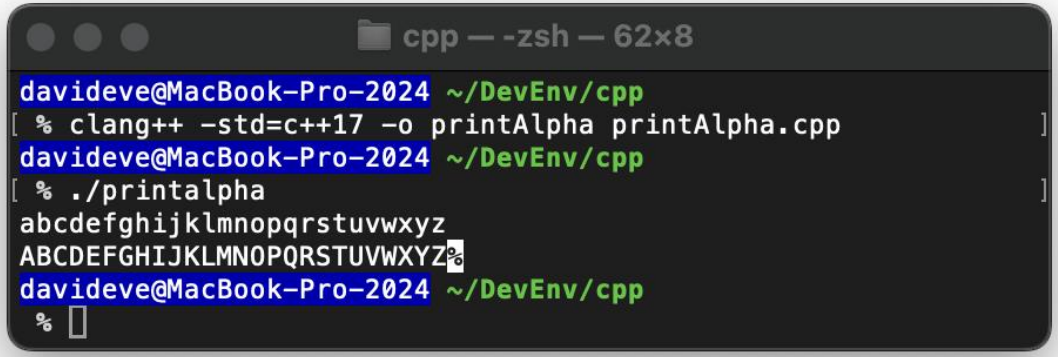
ASCII Control Code Characters (0 – 31)

Control codes			Printable characters								
0d	0x	Code	0d	0x	Char	0d	0x	Char	0d	0x	Char
0	0	NULL	32	20	SPACE	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BELL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	HT	41	29	)	73	49	I	105	69	i
10	0a	LF	42	2a	*	74	4a	J	106	6a	j
11	0b	VT	43	2b	+	75	4b	K	107	6b	k
12	0c	FF	44	2c	,	76	4c	L	108	6c	l
13	0d	CR	45	2d	-	77	4d	M	109	6d	m
14	0e	SO	46	2e	.	78	4e	N	110	6e	n
15	0f	SI	47	2f	/	79	4f	O	111	6f	o
16	10	DLE	48	30	0	80	50	P	112	70	p

Control codes			Printable characters								
0d	0x	Code	0d	0x	Char	0d	0x	Char	0d	0x	Char
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1a	SUB	58	3a	:	90	5a	Z	122	7a	z
27	1b	ESC	59	3b	;	91	5b	[	123	7b	{
28	1c	FS	60	3c	<	92	5c	\	124	7c	|
29	1d	GS	61	3d	=	93	5d	]	125	7d	}
30	1e	RS	62	3e	>	94	5e	^	126	7e	~
31	1f	US	63	3f	?	95	5f	_	127	7f	DEL

printing alphabet in lower and upper case

```cpp
#include <cstdio>

int main() {
  char alphabet[27];
  for (int i = 0; i<26; i++) {
    alphabet[i] = i + 97;
  }
  alphabet[26] = 0;
  printf("%s\n", alphabet);
  for (int i = 0; i<26; i++) {
    alphabet[i] = i + 65;
  }
  printf("%s", alphabet);
}
```



User-Defined Types

- Enumerations: excellent for modeling categorical concepts
- Classes: flexibility to pair data with functions
- Unions: all members share the same memory location

enum class C++

- creates a type-safe, scoped enumeration

```
enum class Race {
    Dinan, Teklan, Ivyn, Moiran,
    Camite, Julian, Aidan
};
```

- Values are internally integers, but using names improves clarity / safety

- Access values using scope resolution

```
Race langobard_race = Race::Aidan;
```

Switch Statements C++

- switch enables multi-way branching based on an int or enum value
- Each case matches a constant expression
- Execution continues until break or end of switch (always use break) or risk fall through
- Use default to handle unmatched values

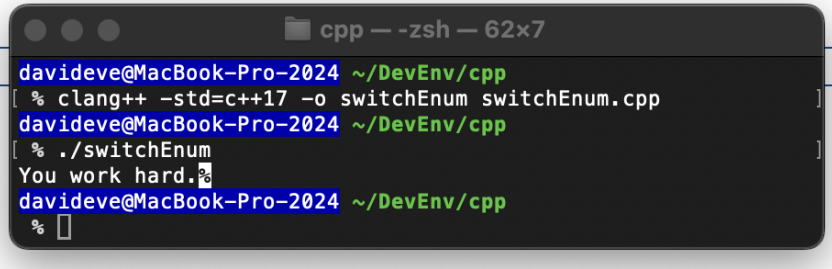
```
switch (value) {
  case optionA:
    // Handle A
    break;
  case optionB:
    // Handle B
    break;
  default:
    // Fallback case
}
```

Using Switch with enum class C++

- enum class is **scoped**: values accessed as Race::Name
- switch branches based on the **value of race**
- break prevents **fall-through**
- default handles unrecognized values (e.g., if enum is extended)

switchEnum.cpp > ...

```cpp
#include <cstdio>

enum class Race {
  Dinan, Teklan, Ivyn, Moiran,
  Camite, Julian, Aidan
};

int main() {
  Race race = Race::Dinan;

  switch (race) {
    case Race::Dinan:  printf("You work hard."); break;
    case Race::Teklan: printf("You are very strong."); break;
    case Race::Ivyn:   printf("You are a great leader."); break;
    case Race::Moiran: printf("My, how versatile you are!"); break;
    case Race::Camite: printf("You're incredibly helpful."); break;
    case Race::Julian: printf("Anything you want!"); break;
    case Race::Aidan:  printf("What an enigma."); break;
    default:           printf("Error: unknown race!");
  }
}
```

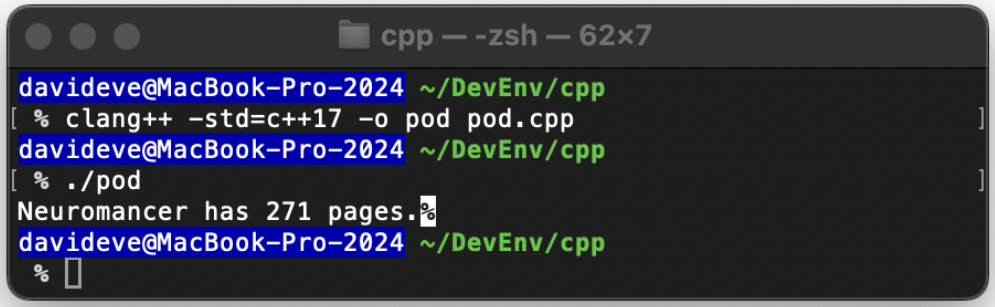


Plain-Old-Data (POD) Classes

- A POD is a simple user-defined type for storing related data
- Defined with struct and made of members (like a heterogeneous array)
- No constructors, destructors, or methods – just data

pod.cpp > ...

```cpp
#include <cstdio>

struct Book {
  char name[256];   // title
  int year;         // publication year
  int pages;        // page count
  bool hardcover;   // binding type
};

int main() {
  Book neuromancer;                  // Declare variable
  neuromancer.pages = 271;           // Set member
  printf("Neuromancer has %d pages.", neuromancer.pages);  // Access member
}
```



Unions

- Similar to a struct, but all members share the same memory location. Only one member valid at any given time.
- Memory-efficient: uses size of the largest member only
- Acts as multiple interpretations of the same memory block
- Often used in low-level programming
  - cross platform
  - interfacing with C libraries
  - bitfield packing (minimize space and supports bitwise logic)

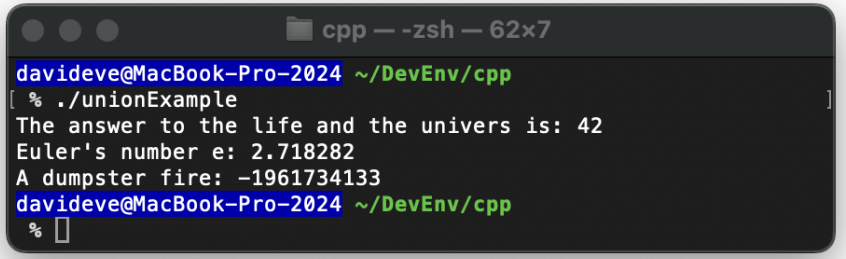
```
union Variant {
  char string[10];
  int integer;
  double floating_point;
};
```

```
union Flags {
  struct {
    unsigned int a : 1;
    unsigned int b : 2;
    unsigned int c : 5;
  };
  uint8_t all;
};
```

Union: Many views, One Memory

unionExample.cpp > ...

```cpp
#include <cstdio>
union Variant {
    char string[10];
    int integer;
    double floating_point;
};


int main(){
    Variant v;
    v.integer = 42;
    printf("The answer to the life and the univers is: %d\n", v.integer);

    v.floating_point = 2.7182818284;
    printf("Euler's number e: %f\n", v.floating_point);

    printf("A dumpster fire: %d\n", v.integer);
}
```



POD Classes

Review:

- PODs only contain data members
- POD: simple and lightweight
- PODs: useful in low-level or data-focused use cases

Limitations:

- PODs: cannot manage complex logic (awkward)

Solution: Encapsulation

- combines data and functions
- provides modularity and clarity

Encapsulation in C++

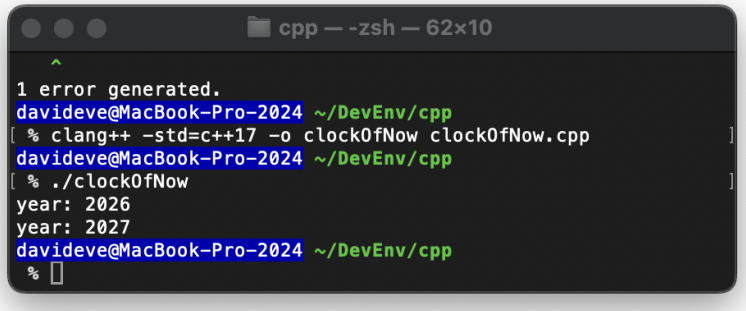
Encapsulation: Binding data with functions

Benefits:

- Cohesion: keeps related code and data in one place (easier to read/ maintain)
- Information Hiding: prevents other parts of program from misusing internal state

clockOfNow.cpp > ...

```cpp
#include <cstdio>

struct ClockOfTheLongNow{
    void add_year(){
        year++;
    }
    int year;
};

int main() {
    ClockOfTheLongNow clock;
    clock.year = 2025;
    clock.add_year();
    printf("year: %d\n", clock.year);
    clock.add_year();
    printf("year: %d\n", clock.year);

};
```

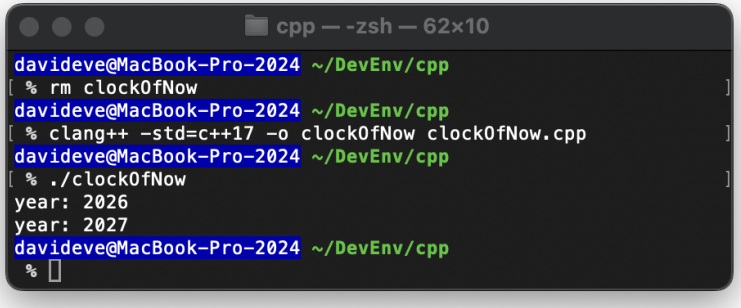


Methods: (member) Functions Inside Classes

- Defined inside a class
- Have full access to all members
- Represent behavior tied to class state

clockOfNow.cpp > ...

```cpp
#include <cstdio>

struct ClockOfTheLongNow {
    ClockOfTheLongNow(int y) { set_year(y); }
    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(2025);
    clock.add_year(); printf("year: %d\n", clock.get_year());
    clock.add_year(); printf("year: %d\n", clock.get_year());
    return 0;
}
```

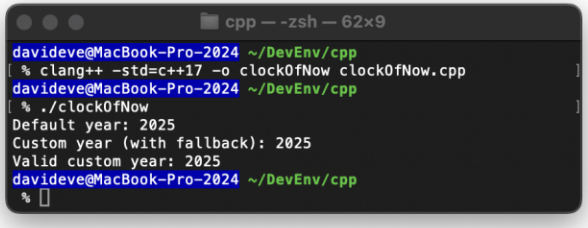


Constructors

- Special function with no return type
- Same name as class
- Runs automatically when object is created

clockOfNow.cpp > ...

```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 default_clock, custom_clock(2020), valid_clock(2025);
    printf("Default year: %d\n", default_clock.get_year());
    printf("Custom year (with fallback): %d\n", custom_clock.get_year());
    printf("Valid custom year: %d\n", valid_clock.get_year());
    return 0;
}

```



Object Initialization in C++

- Initialization = giving an object its first value when created
- Simple rules (be consistent)
  1. int a = 0; // explicit value
  2. Int b{}; //braced initialization (zero)
  3. int c = {}; //equals + braces (zero)

  4. int d; //uninitialized

Initializing POD Structs

```
#include <cstdint>

struct PodStruct {
  uint64_t a;
  char b[256];
  bool c;
};

int main() {
  PodStruct initialized_pod1{};     ❶   // All fields zeroed
  PodStruct initialized_pod2 = {}; ❷   // All fields zeroed

  PodStruct initialized_pod3{ 42, "Hello" }; ❸   // Fields a & b set; c = 0
  PodStruct initialized_pod4{ 42, "Hello", true }; ❹ // All fields set
}
```

Array Initialization

```
int main() {
  int array_1[]{ 1, 2, 3 };  ❶ // Array of length 3; 1, 2, 3
  int array_2[5]{};          ❷ // Array of length 5; 0, 0, 0, 0, 0
  int array_3[5]{ 1, 2, 3 }; ❸ // Array of length 5; 1, 2, 3, 0, 0
  int array_4[5];            ❹ // Array of length 5; uninitialized values
}
```

Constructors and Initialization Full Featured Classes

- Fully featured classes are always initialized using one of their constructors.
- Constructor called depends on the argument provided.

```
#include <cstdio>

struct Taxonomist {
  --snip--
};

int main() {
  Taxonomist t1; ❶
  Taxonomist t2{ 'c' }; ❷
  Taxonomist t3{ 65537 }; ❸
  Taxonomist t4{ 6.02e23f }; ❹
  Taxonomist t5('g'); ❺
  Taxonomist t6 = { 'l' }; ❻
  Taxonomist t7{}; ❼
  Taxonomist t8(); ❽
}
```

```
(no argument) ❶
char: c ❷
int: 65537 ❸
float: 602000017271895229464576.000000 ❹
char: g ❺
char: l ❻
(no argument) ❼
```

Destructor

- A special method called automatically before an object is destroyed
  - Free memory
  - close files
  - flush network buffers
- declared with a tilde ~class name
- Takes no arguments, returns nothing

```
#include <cstdio>
struct Earth {
  ~Earth() { // Earth's destructor
       printf("Making way for hyperspace bypass");
  }
}
```