Basics of Functional and Generic Programming

The Art of Computer Programming

Pierre Senellart

PSL UNIVERSITÉ PARIS

5 January 2026

Outline

Two Kinds of Abstraction

- In imperative programming, we often write code that is:
  - tied to a concrete data type (a Python `list` with only integers, `int[]` in C, `vector<int>` in C++)
  - tied to a concrete behavior (a complex function that relies on one specific transformation, test, or comparison)
- Two orthogonal techniques help us write more reusable code:

  Functional programming (FP) abstract over behavior (functions)
  Generic programming (GP) abstract over types (parametricity)
- These ideas exist in all three languages:

  Python: first-class functions; genericity via duck typing
  C: function pointers; manual genericity via `void *`
  C++: lambdas and algorithms; templates

Motivating Example: Code Reuse

- We want to express reusable patterns:
  - Apply a transformation to every element (map)
  - Keep only elements satisfying a predicate (filter)
  - Aggregate a sequence into one value (reduce/fold)
- And we want these patterns to work:
  - for many behaviors (different transformations / tests)
  - for many types (ints, floats, strings, user-defined types)
- Goal: separate algorithmic structure from details

Motivating Example: Sorting

- Sorting is a generic algorithm:
  - works for many element types (`int`, `std::string`, objects of custom classes...)
- But sorting also depends on a comparison:
  - FP view: pass a comparison function $\text{cmp}(a, b)$
  - GP view: require the element type to provide an ordering (e.g., `operator<` in C++, Python's `__lt__`)
- Two common ways to customize sorting:

  Provide a comparator function (plug behavior in)

  Instantiate a concrete ordered type (bake behavior into the type)

Functional and Generic Programming

	Functional programming	Generic programming
Abstract over	Behavior	Types
Mechanism	Functions as values	Type parameters / type erasure
Typical tool	Higher-order functions	Parametric polymorphism
Pitfalls	Side effects, unclear intent	Unsafe casts, unclear interface

Outline

Outline

Functions as First-Class Values

- A first-class value is something that can be:
  - stored in a variable
  - passed as argument
  - returned as a result of a function
- In functional programming, functions are first-class values
- This enables higher-order functions, typically:
  - map: apply a function to each element
  - filter: keep elements satisfying a condition
  - fold/reduce: combine all elements
- Benefits:
  - reusable control patterns
  - composability (pipelines)
  - clearer separation between what and how

A Generic FP Pattern: Map / Filter / Reduce

- Conceptually:

$$\begin{aligned} \text{Map} \quad & (f, [x_1, \ldots, x_n]) \mapsto [f(x_1), \ldots, f(x_n)] \\ \text{Filter} \quad & (p, [x_1, \ldots, x_n]) \mapsto [x_i \mid p(x_i)] \\ \text{Reduce/fold} \quad & (\oplus, z, [x_1, \ldots, x_n]) \mapsto z \oplus x_1 \oplus \cdots \oplus x_n \end{aligned}$$

- Same pattern, different behaviors (f, p, $\oplus$)
- Same pattern, potentially different element types

Callbacks: Passing Behavior to Another Component

- A callback is a function (or callable object) that you give to another function or library so that it can call you back at the right time
- Intuition: instead of writing the loop yourself, you provide the piece of behavior that should happen inside someone else's loop
- Common situations:
  - Traversal: "do this for each element"
  - Filtering: "keep elements that satisfy this predicate"
  - Sorting: "compare two elements like this"
  - Events: "when the user clicks / when data arrives, run this"
- Callbacks are the simplest and most common form of higher-order programming (functions taking functions)

Outline

Functional Programming in Python: The Essentials

- Functions are objects:
  - assigned to variables
  - passed to other functions
- Anonymous functions: `lambda`
- Built-ins and idioms:
  - `map`, `filter`
  - list comprehensions (often clearer)
  - `functools.reduce`
- Many interfaces are designed around callbacks: sorting key, event handlers, etc.

Python: Higher-Order Functions

Python

```
def apply_twice(f, x):
    return f(f(x))

def inc(x): return x + 1

print(apply_twice(inc, 10))           # 12
print(apply_twice(lambda t: 2*t, 3))  # 12
```

- `f` is a value, passed as a parameter
- `lambda` creates an anonymous function

Anonymous Functions: Lambdas

- A lambda is an anonymous function:
  - it has no name
  - it is usually written where it is used
  - it represents a small piece of behavior
- The name comes from the lambda-calculus, an approach to formalize computation based on logic
- Python:
  - `lambda` creates a function object
  - Syntax is intentionally restricted: one expression only (no statements, no assignment)

Python

```
# lambda expression
square = lambda x: x * x

# equivalent named function
def square(x):
    return x * x
```

Python: Map / Filter in Two Styles

Python

```
xs = [1, 2, 3, 4, 5, 6]

ys = list(map(lambda x: x*x, xs))
zs = list(filter(lambda x: x % 2 == 0, xs))
print(ys, zs)
```

Python

```
ys = [x*x for x in xs]              # map
zs = [x for x in xs if x % 2 == 0] # filter
```

- Comprehensions are often more readable than `map`/`filter`
- Still the same underlying idea: pass behavior as a value

Reducing a Collection in Python

- Reduce (also called fold) combines all elements of a collection into a single value
- It requires:
  - a binary function `f(acc, x)`
  - an initial value (the accumulator)
- In Python, `reduce` is provided by `functools`

Python

```python
from functools import reduce

xs = [1, 2, 3, 4]

# sum = (((0 + 1) + 2) + 3) + 4
s = reduce(lambda acc, x: acc + x, xs, 0)
# product = (((1 * 1) * 2) * 3) * 4
p = reduce(lambda acc, x: acc * x, xs, 1)
```

- Many common reductions have dedicated functions: `sum`, `max`, `min`, `any`, `all`. Prefer these when they express intent more clearly

Sorting in Python with a Callback

- Python sorting (`list.sort`, `sorted`) is:
  - generic: works for many element types
  - customizable: behavior passed as a callback
- The most common customization uses a key function:
  - called once per element
  - transforms elements into values that are compared (not a comparator function)

Python

```
names = ["Alice", "bob", "CHARLES"]

# Sort by lowercase name
names.sort(key=lambda s: s.lower())
print(names)
```

- Conceptually:
  - sorting algorithm controls the loop
  - your `lambda` provides the comparison criterion
- This is a classic example of a callback

Closures in Python: Capturing an Environment

- A closure is a function that:
  - is defined inside another function
  - captures variables from its surrounding scope
- Captured variables are:
  - neither local to the function
  - nor global
  - but remembered by the function object
- Closures allow us to:
  - specialize behavior without defining new global functions
  - carry state together with behavior

Python Closures: A Simple Example

Python

```
def make_adder(k):
    def add(x):
        return x + k    # k is captured
    return add

add10 = make_adder(10)
add3  = make_adder(3)

print(add10(5))  # 15
print(add3(5))   # 8
```

- `add` remembers the value of `k` from its creation
- Each call to `make_adder` creates a new environment
- Functions are both code + captured data

Closures and Callbacks in Python

- Closures are especially useful with callbacks
- They let us customize behavior without changing the interface

Python

```
def make_key(ignore_case):
    return (lambda s: s.lower() if ignore_case else s)

names = ["Alice", "bob", "Charles"]

names.sort(key=make_key(ignore_case=True))
print(names)
```

- The lambda returned by `make_key` is a closure: it captures `ignore_case`
- This pattern is extremely common in functional-style interfaces

Outline

Functional Programming in C: Function Pointers

- C functions are not first-class values, but we can use function pointers
- A function pointer is a pointer to the memory address of a function
- Used for callbacks:
  - sorting (comparison function)
  - traversals (apply a function to each element)
  - library hooks
- Limitations (compared to Python/C++):
  - no closures (no captured environment)
  - awkward syntax
  - manual state passing when needed

C: Function Pointer Syntax

- A function pointer type mirrors a function declaration, with the function name replaced by `(*name)`
- General rule:

C

```
T f(A1, A2, ..., An);          // function declaration
T (*pf)(A1, A2, ..., An);      // pointer to such a function
```

- Parentheses are mandatory:
  - `T *pf(A)` would mean "function returning `T *`"
  - `T (*pf)(A)` means "pointer to function returning `T`"
- Parameter names are optional in pointer types:

C

```
int (*cmp)(const int *, const int *);
```

- Calling syntax:
  - if `pf` is a function pointer, then `pf(x)` calls the function
  - `(*pf)(x)` is equivalent (explicit dereference)
- Reading tip: start at the identifier, then read outward, respecting parentheses

C: Passing a Function as Parameter

C

```c
#include <stdio.h>

int inc(int x) { return x + 1; }

int apply_twice(int (*f)(int), int x) {
  return f(f(x));
}

int main(void) {
  printf("%d\n", apply_twice(inc, 10)); // 12
}
```

- `int (*f)(int)`: pointer to function `int -> int`
- Behavior is now a parameter

C: “Map” Over an Array (Manual)

C

```c
#include <stddef.h>

void map_int(int *out, const int *in, size_t n, int (*f)(int)) {
  for (size_t i = 0; i < n; i++) out[i] = f(in[i]);
}
```

- This is functional programming in spirit: a reusable control pattern
- But it is specialized to `int` (generic programming later)

Outline

Functional Programming in C++: Lambdas and Algorithms

- C++ supports multiple ways to represent callable behavior:
  - function pointers
  - function objects (types with `operator()`)
  - lambdas (often preferred for simple behaviors)
- Standard algorithms expect callables:
  - `std::transform` (map)
  - `std::count_if`, `std::remove_if` (filter-like)
  - `std::accumulate` (reduce)
- Key concept: behavior is passed as a value

C++: Lambda + std::transform

C++

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>

int main() {
  std::vector<int> xs{1,2,3,4,5,6};
  std::vector<int> ys(xs.size());

  std::transform(xs.begin(), xs.end(), ys.begin(),
                 [](int x){ return x*x; });

  int sum = std::accumulate(ys.begin(), ys.end(), 0);
  std::cout << "sum of squares = " << sum << "\n";
}
```

C++: Captures (Closures)

C++

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> xs{1,2,3,4,5,6};
  int threshold = 3;

  int cnt = std::count_if(xs.begin(), xs.end(),
                          [threshold](int x){ return x > threshold; });

  std::cout << cnt << "\n"; // 3 elements: 4,5,6
}
```

- The lambda captures `threshold`: this is what C cannot do directly

C++ Lambdas: Capture Defaults

- In C++, lambdas can capture variables from their surrounding scope
- Capture syntax appears between brackets `[ ... ]`
- Using capture defaults avoids listing variables one by one
- Two main capture defaults:

  `[=]` capture all used variables by value

  `[&]` capture all used variables by reference

C++

```
int factor = 2;
std::vector<int> xs{1,2,3};

std::transform(xs.begin(), xs.end(), xs.begin(),
               [=](int x) { return factor * x; });
```

- Equivalent explicit capture:

C++

```
[factor](int x) { return factor * x; }
```

Outline

Outline

What Is Generic Programming?

- A program is generic if it works for many types:
  - without duplicating the algorithm for each type
  - while preserving correctness and (ideally) safety
- Classic approaches:

  Static generics (C++ templates): checked at compile time, no overhead

  Dynamic generics (Python duck typing): checked at runtime, very flexible

  Manual erasure (C `void *`): powerful but unsafe if misused
- Genericity is about interfaces:
  - What operations do we need? ($\leqslant$, copy, hash, ...)
  - How is that expressed in each language?

Generic Algorithms + Parameters

- Many algorithms have:
  - a type parameter (what kind of elements?)
  - a behavior parameter (how to compare / transform?)
- Example: sorting
  - generic over element type T
  - customizable by comparator $\text{cmp} : T \times T \to \{\text{true}, \text{false}\}$
- This is where functional and generic programming naturally meet

Outline

Generic Programming in Python: Duck Typing

- Many Python functions are implicitly generic: they work for any object that supports the required operations
- Example interfaces
  - "iterable": can be used in `for`
  - "sequence-like": supports `len` and indexing
  - "orderable": supports comparisons
- Strength: high flexibility
- Risk: errors may show up at runtime

Iterables in Python: An Interface

- In Python, an iterable is any object that can be used in:
  - a `for` loop
  - functions like `list`, `sum`, `map`, `sorted`, ...
- This is not enforced by a formal type system: "if it behaves like an iterable, Python accepts it"
- The iterable interface consists of two methods:

  `__iter__()` returns an iterator object

  `__next__()` returns the next element, or raises the `StopIteration` exception
- Most built-in types are iterable: lists, tuples, strings, dictionaries, files, ranges, ...

Defining Your Own Iterable (Explicit Iterator)

Python

```
class CountUpTo:
    def __init__(self, n):
        self._n = n

    def __iter__(self):
        self._current = 0
        return self

    def __next__(self):
        if self._current > self._n:
            raise StopIteration
        x = self._current
        self._current += 1
        return x

for x in CountUpTo(3):
    print(x) # prints: 0 1 2 3
```

Generators and `yield`

- Python provides a simpler way to define iterables: generators
- A generator function:
  - looks like a normal function
  - uses the keyword `yield` instead of `return`
- Meaning of `yield`:
  - produce a value for the iteration
  - pause the function's execution
  - resume later from the same point
- Each call to `yield` corresponds to one step of iteration
- Generators are lazy: values are produced on demand

Defining an Iterable with `yield`

Python

```
def count_up_to(n):
    current = 0
    while current <= n:
        yield current
        current += 1
```

Python

```
for x in count_up_to(3):
    print(x) # prints: 0 1 2 3
```

- Calling `count_up_to(3)` returns a generator object
- The generator:
  - automatically implements `__iter__` and `__next__`
  - raises `StopIteration` when finished
- Preferred style for defining custom iterables in Python

Outline

Generic Programming in C: `void *` and Conventions

- C has no built-in support for generic programming
- Common technique: type erasure with `void *` (pointer to "anything")
  - treat pointers uniformly
  - pass element size explicitly
  - pass callbacks for behavior (comparison, etc.)
- Standard example: `qsort`
- Risks:
  - no compile-time type checking
  - manual type conversions (casts) everywhere
  - easy to get wrong (undefined behavior)

Pointer Casts in C (1/2)

- In C, a cast explicitly converts a value to another type
- General cast syntax:

C

```
(T) expression
```

- Casting value (not function!) pointers:
  - changes the static type of a pointer
  - does not change the underlying address

C

```
void *p = /* some pointer value */;

int *pi = (int *)p;          // cast void* to int*
const char *s = (const char *)p;
```

- In C (but not in C++!), conversions `T* -> void*` and `void* -> T*` are permitted (often without explicit casts), but dereferencing still requires the right type
- Dereferencing requires a cast to the correct pointer type:

C

```
int x = *(int *)p;
```

Pointer Casts in C (2/2)

- Very common pattern in generic C interfaces:

C

```c
int cmp(const void *a, const void *b) {
  int x = *(const int *)a;
  int y = *(const int *)b;
  return (x > y) - (x < y);
}
```

- Important:
  - the cast tells the compiler how to interpret the bytes
  - using the wrong type leads to undefined behavior

C: qsort as Generic Programming

C

```c
#include <stdlib.h>

int cmp_int(const void *a, const void *b) {
  int x = *(const int*)a;
  int y = *(const int*)b;
  return (x > y) - (x < y);
}

int main(void) {
  int xs[] = {4, 1, 3, 2};
  qsort(xs, 4, sizeof(int), cmp_int);
}
```

- Generic over element type: array of bytes + element size
- Behavior parameter: comparator callback

Generic Programming in C: Macros

- Besides `void *`, C offers a limited form of genericity via preprocessor macros
- Idea:
  - Write code once using placeholders
  - Instantiate it by textual substitution before compilation
- This can simulate:
  - generic functions
  - generic data structures
- Typical use cases:
  - type-specific inline functions
  - small, performance-critical abstractions

C Macros: Example of a Generic max

C

```c
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int x = MAX(3, 5);         // int
double y = MAX(2.5, 1.7); // double
```

- The macro works for many types: textual genericity
- But:
  - arguments may be evaluated multiple times
  - no type checking
  - debugging is harder (errors after expansion)
- This is not true parametric polymorphism

C Macros: Generic Code Generation

C

```c
/* Define a type-specific function by macro expansion */
#define DEFINE_SWAP(T)            \
void swap_##T(T *a, T *b) {       \
  T tmp = *a; *a = *b; *b = tmp; \
}

DEFINE_SWAP(int)
DEFINE_SWAP(double)
```

- Macros can generate type-specific functions
- Similar in spirit to C++ templates, but:
  - purely textual substitution
  - no scope, no type checking
  - error messages refer to expanded code
- Useful in low-level libraries, but must be used carefully

Macros vs Other Generic Techniques in C

Technique	Characteristics
Macros	Fast, inline, syntactic genericity; unsafe, hard to debug
`void *` + callbacks	Runtime genericity; flexible but unsafe casts and indirection
Manual duplication	Safe but verbose and error-prone

- Macros are best seen as a last-resort abstraction tool
- Prefer clearer techniques when possible

Outline

Generic Programming in C++: Templates

- C++ templates provide parametric polymorphism:
  - write one algorithm, instantiate for many types
  - type-checked at compile time
  - usually zero runtime overhead
- STL (C++'s standard template library) is built around templates:
  - containers: `std::vector<T>`
  - algorithms: `std::sort`, `std::transform`, ...
  - iterators: generic interface to sequences

C++: A Simple Function Template

C++

```cpp
template<typename T>
T my_max(T a, T b) {
  return (a < b) ? b : a;
}

int main() {
  auto x = my_max(3, 7);          // T = int
  auto y = my_max(2.0, 1.5);      // T = double
}
```

- Works for any `T` supporting `<`
- The constraint exists even if not explicitly stated

C++: Generic Sorting with a Generic Lambda

C++

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> xs{4, 1, 3, 2};
  std::sort(xs.begin(), xs.end(),
            [](const auto& a, const auto& b) {
              return a > b;   // generic comparison
            });
  for (const auto& x : xs) std::cout << x << " ";
}
```

- The lambda uses `auto` parameters: it is generic (templated by the compiler); it works for any type supporting `operator>`
- `std::sort` is generic over: the element type (templates) and the comparison behavior (callback)

Generic Classes in C++: Templates

- In C++, class templates allow defining data structures independently of the element type
- Syntax:

C++

```
template<typename T>
class Box {
  T value;
public:
  Box(T v) : value(v) {}
  T get() const { return value; }
};
```

- T is a type parameter
- The class is instantiated for concrete types:
  - Box<int>
  - Box<double>
  - Box<std::string>

Using a Generic Class

C++

```
#include <iostream>
#include <string>

int main() {
  Box<int> bi(42);
  Box<std::string> bs("hello");

  std::cout << bi.get() << "\n";
  std::cout << bs.get() << "\n";
}
```

- The same class definition is reused for many types
- Type checking happens at compile time
- No runtime overhead compared to handwritten versions

Class Templates and Header Files

- Unlike ordinary classes, class templates must be fully visible at the point of use
- Reason:
  - templates are instantiated at compile time
  - the compiler needs the full definition to generate code
- Consequence:
  - method definitions of class templates cannot usually go in a separately compiled `.cpp` file
  - they must be placed in a header file (often with the extension `.hpp` to distinguish from standard `.h` header files)
  - this header is #included wherever the template is used

Class Templates and Header Files: Example

C++

```
#ifndef BOX_HPP
#define BOX_HPP
template<typename T>
class Box {
public:
  T get() const;
private:
  T value;
};

template<typename T>
T Box<T>::get() const {
  return value;
}
#endif /* BOX_HPP */
```

Templates and Inheritance: Caveats

- Combining class templates and inheritance is possible, but subtle
- Key caveats:
  - Base classes may depend on template parameters
  - Members of dependent base classes are not found by unqualified lookup; use `this->`, `Base<T>::`, or a `using` declaration
  - Virtual functions and templates solve different problems
- Design guideline:
  - use templates for *compile-time* polymorphism
  - use inheritance + virtual functions for *runtime* polymorphism
  - mixing both should be done deliberately and sparingly

Templates and Inheritance: Example (1/4)

C++

```
#ifndef MATRIX_HPP
#define MATRIX_HPP
#include <vector>
#include <iostream>

template<typename T>
class Matrix {
std::vector<std::vector<T> > lines;
public:
Matrix(int n, int m, T v) {
  lines.resize(n);
  for(auto &l : lines) {
    l.resize(m);
    for(auto &c : l) c=v;
  }
}
// continued next slide
```

Templates and Inheritance: Example (2/4)

C++

```
// continued from previous slide

void print() const
{
  for(const auto &l : lines) {
    for(const auto &c: l) std::cout << c << " ";
    std::cout << "\n";
  }
}
};
#endif /* MATRIX_HPP */
```

Templates and Inheritance: Example (3/4)

C++

```
#ifndef SQUAREMATRIX_HPP
#define SQUAREMATRIX_HPP
#include "Matrix.hpp"

template<typename T>
class SquareMatrix : public Matrix<T> {
public:
SquareMatrix(int n, T v) : Matrix<T>(n, n, v) {
}

void printSquared() const {
  this->print(); // print(); does not work
}
};
#endif /* SQUAREMATRIX_HPP */
```

Templates and Inheritance: Example (4/4)

C++

```
// main.cpp
#include "SquareMatrix.hpp"
#include "Matrix.hpp"

int main()
{
  SquareMatrix<double> dm(3, 4.);
  dm.print();
  dm.printSquared();
  Matrix<int> im(4, 3, 42);
  im.print();
}
```

Outline

One Pattern, Three Languages

- Goal: reusable algorithmic skeleton + pluggable details
- Functional programming view: pass behavior as a parameter
- Generic programming view: keep code valid for many types
- Typical combined example:
  - sort a list/array/vector using a custom comparison
  - transform a container using a custom function
- Language takeaways:

  Python: very easy functional programming; genericity via duck typing (optionally documented with type hints)

  C: functional programming via function pointers; generic programming via `void *` + conventions

  C++: functional programming via lambdas; generic programming via templates (STL as the model)

Common Pitfalls

- Functional programming pitfalls:
  - mixing computation with side effects in callbacks
  - overusing anonymous functions when naming helps readability
- Generic programming pitfalls:
  - unclear interfaces (what operations are required?)
  - in C: unsafe casts / wrong sizes / lifetime bugs
  - in C++: template errors that hide the real problem, complex combinations of templates and inheritance
- Best practice: make interfaces (which behavior a function or a class needs to provide) explicit (documentation, naming, tests)

Summary

- Two orthogonal abstractions:
  - Functional programming: functions as values; higher-order functions; lambdas
  - Generic programming: code independent of concrete types; interfaces over operations
- In Python:
  - Functional programming is natural; generic programming is implicit (duck typing) but should be documented
- In C:
  - Functional programming via function pointers; generic programming via `void *` (powerful but unsafe and a bit cumbersome)
- In C++:
  - Functional programming via lambdas/algorithms; generic programming via templates (STL is the reference design)
- Functional + Generic programming together = reusable algorithms with customizable behavior

Licensing