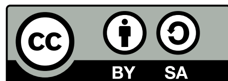
# Structured Programming

## The Art of Computer Programming

Pierre Senellart




22 September 2025

# Outline

Structured Programming

Control Flow Statements

Functions

## Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

  **Imperative** the program is a sequence of instructions executed one after another

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

  **Imperative** the program is a sequence of instructions executed one after another

  **Functional** the program is a collection of mathematical functions

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

| | |
|---|---|
| Imperative | the program is a sequence of instructions executed one after another |
| Functional | the program is a collection of mathematical functions |
| Object-oriented | the program is described as a collection of objects, with properties and methods |

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

| | |
|---|---|
| Imperative | the program is a sequence of instructions executed one after another |
| Functional | the program is a collection of mathematical functions |
| Object-oriented | the program is described as a collection of objects, with properties and methods |
| Generic | the program is written with type-agnostic algorithms that can be instantiated for many data types |

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

Imperative: the program is a sequence of instructions executed one after another

Functional: the program is a collection of mathematical functions

Object-oriented: the program is described as a collection of objects, with properties and methods

Generic: the program is written with type-agnostic algorithms that can be instantiated for many data types

Logic: the program is a combination of logical reasoning rules

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

| | |
|---|---|
| Imperative | the program is a sequence of instructions executed one after another |
| Functional | the program is a collection of mathematical functions |
| Object-oriented | the program is described as a collection of objects, with properties and methods |
| Generic | the program is written with type-agnostic algorithms that can be instantiated for many data types |
| Logic | the program is a combination of logical reasoning rules |
| Event-driven | the program specifies how to respond to events |

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

| | |
|---|---|
| Imperative | the program is a sequence of instructions executed one after another |
| Functional | the program is a collection of mathematical functions |
| Object-oriented | the program is described as a collection of objects, with properties and methods |
| Generic | the program is written with type-agnostic algorithms that can be instantiated for many data types |
| Logic | the program is a combination of logical reasoning rules |
| Event-driven | the program specifies how to respond to events |
| Synchronous | the program describes, at each clock cycle, how the output data evolves as a function of the input data |

# Programming Paradigms

- Style of approach to programming: how to express the solution to a problem as a computer program
- Different programming paradigms:

| | |
|---|---|
| Imperative | the program is a sequence of instructions executed one after another |
| Functional | the program is a collection of mathematical functions |
| Object-oriented | the program is described as a collection of objects, with properties and methods |
| Generic | the program is written with type-agnostic algorithms that can be instantiated for many data types |
| Logic | the program is a combination of logical reasoning rules |
| Event-driven | the program specifies how to respond to events |
| Synchronous | the program describes, at each clock cycle, how the output data evolves as a function of the input data |
| ... | |

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

  Imperative C, Pascal, FORTRAN 77

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

  **Imperative** C, Pascal, FORTRAN 77
  **Functional** Haskell

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

| | |
|---|---|
| Imperative | C, Pascal, FORTRAN 77 |
| Functional | Haskell |
| Object-oriented | Smalltalk, Eiffel |

## Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

  | | |
  |---|---|
  | Imperative | C, Pascal, FORTRAN 77 |
  | Functional | Haskell |
  | Object-oriented | Smalltalk, Eiffel |
  | Logic | Prolog |

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

| | |
|---|---|
| Imperative | C, Pascal, FORTRAN 77 |
| Functional | Haskell |
| Object-oriented | Smalltalk, Eiffel |
| Logic | Prolog |
| Event-driven | Visual Basic |

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

| | |
|---|---|
| Imperative | C, Pascal, FORTRAN 77 |
| Functional | Haskell |
| Object-oriented | Smalltalk, Eiffel |
| Logic | Prolog |
| Event-driven | Visual Basic |
| Synchronous | Lustre |

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

| | |
|---|---|
| Imperative | C, Pascal, FORTRAN 77 |
| Functional | Haskell |
| Object-oriented | Smalltalk, Eiffel |
| Logic | Prolog |
| Event-driven | Visual Basic |
| Synchronous | Lustre |

- But most languages mix and encourage multiple paradigms of programming

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

| | |
|---|---|
| Imperative | C, Pascal, FORTRAN 77 |
| Functional | Haskell |
| Object-oriented | Smalltalk, Eiffel |
| Logic | Prolog |
| Event-driven | Visual Basic |
| Synchronous | Lustre |

- But most languages mix and encourage multiple paradigms of programming
- Python: mainly imperative, with major aspects of functional and object-oriented – specific extensions or interfaces use event-driven, reactive, logic programming, . . .

# Paradigms and Programming Languages

- Some languages are dedicated to, or particularly well-suited for, specific programming paradigms, for example:

| | |
|---|---|
| Imperative | C, Pascal, FORTRAN 77 |
| Functional | Haskell |
| Object-oriented | Smalltalk, Eiffel |
| Logic | Prolog |
| Event-driven | Visual Basic |
| Synchronous | Lustre |

- But most languages mix and encourage multiple paradigms of programming
- Python: mainly imperative, with major aspects of functional and object-oriented – specific extensions or interfaces use event-driven, reactive, logic programming, . . .
- C++: primarily imperative and object-oriented, with strong support for generic programming and increasing emphasis on functional style

# Branching and Iteration

- Any form of programming requires two fundamental mechanisms:

## Branching and Iteration

- Any form of programming requires two fundamental mechanisms:

  Branching the ability to make choices

# Branching and Iteration

- Any form of programming requires two fundamental mechanisms:
  - Branching: the ability to make choices
  - Iteration: the ability to repeat actions

# Branching and Iteration

- Any form of programming requires two fundamental mechanisms:

  **Branching** the ability to make choices

  **Iteration** the ability to repeat actions

- Without these, an (imperative) program can only execute instructions in a fixed, linear order

## Branching and Iteration

- Any form of programming requires two fundamental mechanisms:

  Branching the ability to make choices

  Iteration the ability to repeat actions

- Without these, an (imperative) program can only execute instructions in a fixed, linear order
- Early versions of programming languages (FORTRAN, COBOL, BASIC) relied on tests combined with goto instructions: jump to some line of the program if some condition is satisfied

# Branching and Iteration

- Any form of programming requires two fundamental mechanisms:

  Branching the ability to make choices

  Iteration the ability to repeat actions

- Without these, an (imperative) program can only execute instructions in a fixed, linear order
- Early versions of programming languages (FORTRAN, COBOL, BASIC) relied on tests combined with goto instructions: jump to some line of the program if some condition is satisfied
- Most assembly and bytecode are still like this!

## Branching and Iteration

- Any form of programming requires two fundamental mechanisms:
  - Branching the ability to make choices
  - Iteration the ability to repeat actions
- Without these, an (imperative) program can only execute instructions in a fixed, linear order
- Early versions of programming languages (FORTRAN, COBOL, BASIC) relied on tests combined with goto instructions: jump to some line of the program if some condition is satisfied
- Most assembly and bytecode are still like this!
- In the 1960s, the importance of explicit, higher-level constructs (if/else, while, for, functions) to organize (mostly imperative) code was recognized (see [Dijkstra, 1968])

# Branching and Iteration

- Any form of programming requires two fundamental mechanisms:
  - Branching the ability to make choices
  - Iteration the ability to repeat actions
- Without these, an (imperative) program can only execute instructions in a fixed, linear order
- Early versions of programming languages (FORTRAN, COBOL, BASIC) relied on tests combined with goto instructions: jump to some line of the program if some condition is satisfied
- Most assembly and bytecode are still like this!
- In the 1960s, the importance of explicit, higher-level constructs (if/else, while, for, functions) to organize (mostly imperative) code was recognized (see [Dijkstra, 1968])
- Major influence of ALGOL, a language from the late 1950s and early 1960s that introduced the notion of code blocks to group statements together

# Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:

## Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:

  Sequence executing instructions one after another

# Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:

  **Sequence** executing instructions one after another

  **Tests** making decisions (`if`, `switch`/`match`)

# Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:

  **Sequence** executing instructions one after another

  **Tests** making decisions (`if`, `switch`/`match`)

  **Loops** repeating actions (`for`, `while`)

# Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:

  **Sequence** executing instructions one after another

  **Tests** making decisions (`if`, `switch`/`match`)

  **Loops** repeating actions (`for`, `while`)

  **Functions** grouping instructions into reusable, named units

# Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:

  Sequence executing instructions one after another

  Tests making decisions (`if`, `switch`/`match`)

  Loops repeating actions (`for`, `while`)

  Functions grouping instructions into reusable, named units

- Control flow statements: Tests + Loops

# Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:
  - **Sequence** executing instructions one after another
  - **Tests** making decisions (`if`, `switch`/`match`)
  - **Loops** repeating actions (`for`, `while`)
  - **Functions** grouping instructions into reusable, named units
- Control flow statements: Tests + Loops
- The goal is to write programs that are readable, maintainable, and correct without relying on arbitrary jumps (`goto`)

# Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:
  - **Sequence** executing instructions one after another
  - **Tests** making decisions (`if`, `switch`/`match`)
  - **Loops** repeating actions (`for`, `while`)
  - **Functions** grouping instructions into reusable, named units
- Control flow statements: Tests + Loops
- The goal is to write programs that are readable, maintainable, and correct without relying on arbitrary jumps (`goto`)
- Structured programming became the standard approach in most imperative languages starting in the 1970s; by imperative programming, nowadays people mean structured programming

## Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:
  - Sequence: executing instructions one after another
  - Tests: making decisions (`if`, `switch`/`match`)
  - Loops: repeating actions (`for`, `while`)
  - Functions: grouping instructions into reusable, named units
- Control flow statements: Tests + Loops
- The goal is to write programs that are readable, maintainable, and correct without relying on arbitrary jumps (`goto`)
- Structured programming became the standard approach in most imperative languages starting in the 1970s; by imperative programming, nowadays people mean structured programming
- All three languages we will use – Python, C, and C++ – rely on structured programming constructs

## Structured Programming

- Structured programming is a programming paradigm that organizes code using four fundamental constructs:
  - **Sequence** executing instructions one after another
  - **Tests** making decisions (`if`, `switch`/`match`)
  - **Loops** repeating actions (`for`, `while`)
  - **Functions** grouping instructions into reusable, named units
- Control flow statements: Tests + Loops
- The goal is to write programs that are readable, maintainable, and correct without relying on arbitrary jumps (`goto`)
- Structured programming became the standard approach in most imperative languages starting in the 1970s; by imperative programming, nowadays people mean structured programming
- All three languages we will use – Python, C, and C++ – rely on structured programming constructs
- Other modern languages (Java, C#, Rust, Go, etc.) also follow this paradigm

# Pseudo-code

- An informal language for writing algorithms
- Mix of natural language and structured instructions
- Ignores details of specific programming languages,
- Usually ignores data types
- Expressions often use mathematical notations
- Variable assignment is denoted $var := value$ or $var \leftarrow value$

## Example

**Input:** Integers $a$, $b$, $c$
**Output:** Discriminant of quadratic equation $ax^2 + bx + c = 0$

$\Delta \leftarrow b^2 - 4ac$
**return** $\Delta$

# Outline

Structured Programming

Control Flow Statements

Functions

# Code Blocks

- Code blocks group multiple instructions together to form a single logical unit
- Building block of control flow statements and functions
- Code blocks can be nested

C/C++ Code blocks are delimited by curly braces { }

Python Code blocks are indicated by indentation, the amount of spaces at the beginning of the line – the more spaces the more nested

C

```c
if (x > 0) {
  printf("Positive\n");
  x = x - 1;
}
```

Python

```python
if x > 0:
    print("Positive")
    x = x - 1
```

# A Note on Indenting

- Indenting means adding horizontal space at the beginning of a line of code
- In all programming languages, indentation is not just aesthetic: it helps make the structure of the code clearer
- Text programming editors and integrated development environments help with keeping indenting consistent
- Python: indentation is syntactically significant – it defines code blocks.
  - Usually (style guides for Python [van Rossum et al., 2001]) 4 spaces per indentation level
- C/C++: indentation is not required syntactically, but strongly recommended for readability
  - 2 or 4 spaces per indentation level are common, be consistent!
- Proper indentation helps you and others understand the structure at a glance, even if curly braces exist

# Syntax of Control Flow Statements

- Control flow statements determine the execution order of instructions.
- C/C++:
  - Conditions in parentheses `(condition)`
  - Followed by a code block enclosed in curly braces `{ }`, or (more rarely) by a single line
- Python:
  - No parentheses necessary around the condition
  - Colon `:` at the end of the line
  - Followed by an indented code block
  - Use `pass` for an empty code block (rare)

# Tests

- A test evaluates some expression and executes code depending on its results
- Two common forms of tests:

  If / Else If / Else
  - The expression is a Boolean condition, the If part is executed if true, the optional Else part is executed if false
  - Possibility of chaining them with an optional Else If construct
  - Executes one block among several alternatives depending on conditions
  - C/C++: `if` / `else if` / `else`
  - Python: `if` / `elif` / `else`

  Switch / Match
  - The expression is evaluated and the code for the corresponding case is executed; optional default case
  - C / C++: `switch` (only for integers or byte values); `break` required after each case!
  - Python (3.10+): `match` (for any comparable values)

# Example If Tests in Pseudo-Code

$x \leftarrow 5$
**if** $x > 0$ **then**
    print "x is positive"
**end if**

# Example If Tests in Pseudo-Code

$x \leftarrow 5$
**if** $x > 0$ **then**
  print "x is positive"
**end if**

$x \leftarrow -3$
**if** $x > 0$ **then**
  print "x is positive"
**else**
  print "x isn't positive"
**end if**

# Example If Tests in Pseudo-Code

$x \leftarrow 5$
**if** $x > 0$ **then**
  print "x is positive"
**end if**

$x \leftarrow -3$
**if** $x > 0$ **then**
  print "x is positive"
**else**
  print "x isn't positive"
**end if**

$x \leftarrow 0$
**if** $x > 0$ **then**
  print "x is positive"
**else if** $x < 0$ **then**
  print "x is negative"
**else**
  print "x is zero"
**end if**

# Example If Tests in C/C++

C

```c
int x = 5;
if (x > 0) {
  printf("x is positive\n");
}
```

# Example If Tests in C/C++

C

```
int x = 5;
if (x > 0) {
  printf("x is positive\n");
}
```

C

```
int x = -3;
if (x > 0) {
  printf("x is positive\n");
} else {
  printf("x isn't positive\n");
}
```

# Example If Tests in C/C++

C

```
int x = 5;
if (x > 0) {
  printf("x is positive\n");
}
```

C

```
int x = -3;
if (x > 0) {
  printf("x is positive\n");
} else {
  printf("x isn't positive\n");
}
```

C

```
int x = 0;
if (x > 0) {
  printf("x is positive\n");
} else if (x < 0) {
  printf("x is negative\n");
} else {
  printf("x is zero\n");
}
```

# Example If Tests in Python

Python

```python
x = 5
if x > 0:
    print("x is positive")
```

# Example If Tests in Python

Python

```python
x = 5
if x > 0:
    print("x is positive")
```

Python

```python
x = -3
if x > 0:
    print("x is positive")
else:
    print("x isn't positive")
```

# Example If Tests in Python

Python

```python
x = 5
if x > 0:
    print("x is positive")
```

Python

```python
x = -3
if x > 0:
    print("x is positive")
else:
    print("x isn't positive")
```

Python

```python
x = 0
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```

# Example Switch in C/C++

C

```
int x = 2;
switch (x) {
  case 1: {
    printf("x is one\n");
    break;
  }
  case 2: {
    printf("x is two\n");
    break;
  }
  case 3: {
    printf("x is three\n");
    break;
  }
  default: {
    printf("x is something else\n");
  }
}
```

# Example Match in Python (3.10+)

Python

```
x = 2
match x:
    case 1:
        print("x is one")
    case 2:
        print("x is two")
    case 3:
        print("x is three")
    case _:
        print("x is something else")
```

# Loops

- A loop repeats code while some condition is true or over a range of values
- Common forms of loops:

  While / Do While

  - The code block is executed repeatedly as long as the Boolean condition is true
  - While: condition tested before executing the block
  - Do While: condition tested after executing the block at least once
  - C/C++: `while(condition) { ... }` or `do { ... } while(condition);`
  - Python: `while condition:`

  For

  - Repeats code over a range of values
  - C/C++: `for(initialization; condition; update) { ... }`
  - Python: `for var in range(0, n):`
- In all three languages:
  - `break`: exit the loop immediately
  - `continue`: skip to the next iteration

# Example Loops in Pseudo-Code

$i \leftarrow 1$
**while** $i \leqslant 5$ **do**
  print $i$
  $i \leftarrow i + 1$
**end while**

# Example Loops in Pseudo-Code

$i \leftarrow 1$
**while** $i \leqslant 5$ **do**
  print $i$
  $i \leftarrow i + 1$
**end while**

$i \leftarrow 0$
**repeat**
  $i \leftarrow i + 1$
  print $i$
**until** $i \geqslant 5$

# Example Loops in Pseudo-Code

$i \leftarrow 1$
**while** $i \leqslant 5$ **do**
  print $i$
  $i \leftarrow i + 1$
**end while**

$i \leftarrow 0$
**repeat**
  $i \leftarrow i + 1$
  print $i$
**until** $i \geqslant 5$

**for** $i \leftarrow 1$ **to** 5 **do**
  print $i$
**end for**

# Example Loops in C/C++

C

```
int i = 1;
while (i <= 5) {
  printf("%d\n", i);
  i++;
}
```

# Example Loops in C/C++

C

```c
int i = 1;
while (i <= 5) {
  printf("%d\n", i);
  i++;
}
```

C

```c
int i = 0;
do {
  i++;
  printf("%d\n", i);
} while (i < 5);
```

# Example Loops in C/C++

C

```c
int i = 1;
while (i <= 5) {
  printf("%d\n", i);
  i++;
}
```

C

```c
for (int i = 1; i <= 5; i++) {
  printf("%d\n", i);
}
```

C

```c
int i = 0;
do {
  i++;
  printf("%d\n", i);
} while (i < 5);
```

# Example Loops in Python

**Python**

```python
i = 1
while i <= 5:
    print(i)
    i += 1
```

# Example Loops in Python

Python

```python
i = 1
while i <= 5:
    print(i)
    i += 1
```

Python

```python
# Do While loops do not exist
# in Python, but we can
# simulate them with break
i = 0
while True:
    i += 1
    print(i)
    if (i >= 5):
        break
```

# Example Loops in Python

Python

```python
i = 1
while i <= 5:
    print(i)
    i += 1
```

Python

```python
# Do While loops do not exist
# in Python, but we can
# simulate them with break
i = 0
while True:
    i += 1
    print(i)
    if (i >= 5):
        break
```

Python

```python
for i in range(1, 6):
    print(i)
```

# Shortcut Assignment Operators

- Shortcut assignment operators combine an operation and assignment
- General form:

$$x \text{ op= } y \quad \text{means} \quad x \leftarrow x \text{ op } y$$

- Common operators:
  - `+=`, `-=`
  - `*=`, `/=`
  - `//=` (Python only)
- In C/C++ but not in Python, one can also write:
  - `i++` or `++i` for `i += 1`
  - `i--` or `--i` for `i -= 1`

# Range in Python

- range(start, stop, step) generates a sequence of integers from start (included) to stop (excluded) with step added at each step

Python

```
for i in range(1, 6):
    print(i)  # prints 1, 2, 3, 4, 5

for i in range(2, 10, 2):
    print(i)  # prints 2, 4, 6, 8

for i in range(5, 0, -1):
    print(i)  # prints 5, 4, 3, 2, 1
```

# Outline

Structured Programming

Control Flow Statements

Functions

# Functions

- A function is a named block of code that can be executed (called) from other parts of a program
- It may take parameters (variables local to the function) which receive values called arguments when the function is called
- A function may compute and provide a return value, or return nothing
- To use a function, one performs a function call, giving the function name and (optional) arguments
- Functions allow:
  - Reuse of code
  - Decomposition of a program into smaller, logical units
  - Abstraction: using functions without knowing their internal details
- Good practice: whenever you have a block of code that becomes too long, or that is re-used in multiple places, make a function out of it!

# Functions in Pseudo-Code

**function** $square(\mathrm{x})$
**return** x * x
**end function**

$y \leftarrow square(5)$
print $y$

# Functions in Python

- A function definition uses the keyword `def` and
  - A name
  - A list of parameters (optional; can be empty)
- The function body is defined by indentation (no braces), after a colon `:`, like any block
- The `return` statement provides the return value (or a special `None` value if omitted)

Python

```
def square(x):
    return x * x

y = square(5)
print(y)   # prints 25
```

# Functions in C/C++

- A function definition specifies:
  - A return type (or `void` for no return value)
  - A name
  - Within parameters, a list of parameters with explicit types and names (if no parameters, use `void` in C, and nothing at all in C++)
- The function body is enclosed in { }, like any block
- The `return` statement provides the return value

C

```
int square(int x) {
  return x * x;
}

int main(void) {
  int y = square(5);
  printf("%d", y);    // prints 25
  return 0;
}
```

# Parameter Passing Modes (1/2)

- When calling a function, arguments to the function call become parameters within the function
- The way this is done is called parameter-passing mode and depends on the programming language:

C
- by value: the value of the argument is copied into the parameter variable; modifications to the parameter within the function are not reflected out of the function
- by address: instead of a function taking as parameter a regular value, it can take an address; modification to the value pointed to by this address are reflected out of the function

C++
- by value like in C
- by address like in C
- by reference if the type of the parameter is followed by an & character; modifications of the parameter within the function are reflected out of the function

# Parameter Passing Modes (2/2)

Python

- by object reference: more complex and less intuitive
- The parameter within the function has the same type and address as the argument
- But any assignment of this variable to a new value changes its address, so modifications through assignments are not reflected out of the function
- However, if the parameter has a complex type with update methods (e.g., lists, dictionaries, sets – see next lecture), any modification through update methods are reflected outside

# Pass by Value (C)

C

```c
#include <stdio.h>

void add_one(int x) {
  x = x + 1;
}

int main(void) {
  int a = 5;
  add_one(a);
  printf("%d\n", a); // still 5
  return 0;
}
```

# Pass by Address (C)

C

```
#include <stdio.h>

void add_one(int *x) {
  *x = *x + 1;
}

int main(void) {
  int a = 5;
  add_one(&a);
  printf("%d\n", a); // now 6
  return 0;
}
```

# Pass by Reference (C++)

C++

```
#include <iostream>

void add_one(int &x) {
  x = x + 1;
}

int main() {
  int a = 5;
  add_one(a);
  std::cout << a << std::endl; // now 6
  return 0;
}
```

# Pass by Object Reference (Python) with Re-Assignment

Python

```
def add_one(x):
    x = x + 1   # assignment makes x point to new object

a = 5
add_one(a)
print(a)   # still 5
```

# Pass by Object Reference (Python) with Update

**Python**

```python
def append_one(l):
    l.append(1)   # modify through update method

a = []
append_one(a)
print(a)   # [1]
```

# Scope and Extent of a Local Variable

- The scope of a variable: where in the program the variable can be accessed
- The extent (lifetime) of a variable: how long the variable exists in memory
- Local variables:

  C/C++
  - A variable declared inside a block { ... } is local to that block
  - It cannot be accessed outside the block
  - Its extent lasts from entry into the block until exit
  - Variables declared in nested blocks or inner scopes shadow variables in outer scopes with the same name (but bad idea to have the same name)

  Python
  - A variable assigned inside a function is local to that function
  - Cannot be accessed outside the function
  - Extent lasts from function call until function returns
- Best practice: keep variables as local as possible

# Illustrating Variable Scope

C

```c
{
  ...
  {
    int x = 5; // local to this block
    printf("%d\n", x);
  }
  // x is not accessible here
}
```

Python

```python
def f():
    x = 5  # local to f
    print(x)

# x is not accessible here
```

# Nested Local Scopes

**C**

```c
void f() {
  int x = 1; // outer x
  {
    int x = 2; // different x
    printf("%d\n", x);
    // prints 2
  }
  printf("%d\n", x);
  // prints 1
}
```

**Python**

```python
def f():
    x = 1  # local to f
    if True:
        x = 2  # same x
        print(x)  # prints 2
    print(x)  # prints 2
```

## Call Stack

- Local variables in functions are allocated on the call stack.
- The stack is a region of memory (relatively limited in size) that stores:
  - Local variables of active functions
  - Return addresses for function calls
  - Function parameters (in C/C++)
- Stack allocation is automatic:
  - Memory is reserved when a function is called
  - Memory is freed when the function returns

# Recursion

- Recursion is when a function calls itself, directly or indirectly
- Each recursive call has its own set of local variables
- Two essential components of a recursive function:
    - Base case: the condition under which the recursion stops
    - Recursive case: the part where the function calls itself
- Recursion is often used to solve problems that can be divided into smaller, similar subproblems
- Similar to definition by induction in mathematics
- Limitation: the call stack is often much smaller than the total available of memory available, so deeply nested recursions are not advisable

# Recursive Function Example

**Python**

Python

```python
def fact(n):
    if n <= 1:
        return 1
    return n * fact(n-1)
```

**C**

C

```c
int fact(int n) {
  if (n <= 1) return 1;
  return n * fact(n-1);
}
```

# Iteration vs Recursion

- Both iteration and recursion allow repeating a computation multiple times
- Iteration:
  - Uses loops (`for`, `while`, ...)
  - Variables are updated explicitly
  - May be slightly faster, no need to use the call stack
- Recursion:
  - Function calls itself
  - Each call has its own local variables
  - Often useful to simplify problems that are naturally recursive

# Iteration vs Recursion: Factorial

**Python**

```python
# Iterative factorial
def fact_iter(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

# Recursive factorial
def fact_rec(n):
    if n <= 1:
        return 1
    else:
        return n*fact_rec(n-1)
```

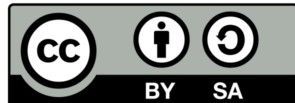
**C**

```c
// Iterative factorial
int fact_iter(int n) {
  int result = 1;
  for(int i=1; i<=n; i++)
    result *= i;
  return result;
}

// Recursive factorial
int fact_rec(int n) {
  if(n <= 1)
    return 1;
  else
    return n * fact_rec(n-1);
}
```

# Licensing

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.



# Bibliography I

Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968. doi: 10.1145/362929.362947. URL `https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf`.

Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. Pep 8 – style guide for python code. `https://peps.python.org/pep-0008/`, 2001.