Control flow in C++

Programação (L.EIC009)

José Proença (FCUP) & João Bispo (FEUP) – *slides by Eduardo R. B. Marques, FCUP*

Outline

- **Function calls**: call and return, call-by-value.
- **Choice instructions**: `if`, `if-else`, `switch-case`
- **Loop instructions**: `while`, `for`, `do-while`
- **More on functions**: function overloading, default values for parameters.
- **Reference variables**, call-by-reference.

Function calls

Example - factorial calculation

```
#include <iostream>
int factorial(int n) {
  int r = 1;
  for (int i = 1; i <= n; i++) r = r * i;
  return r;
}
int main() {
  int n;
  std::cout << "n ? "; std::cin >> n;              // ask "n"
  std::cout <<  n << "! = " << factorial(n) << "\n"; // print "n!"
  return 0;
}
```

Terminology introduced previously

```
int factorial(int n) { // <- function declaration
  // -> function body
  . . .
}
int main() {
  . . .
}
```

From the declarations (also designated as prototypes or signatures):

- `factorial` returns `int` values, and takes one `int` argument
- `main` returns `int` values, and takes no arguments.

Function call and return

::: columns :::: {.column width=45%}

```
int factorial(int n) {
  . . .
  return r; // function return
}
int main() {
  . . .
  // Call to factorial
  . . . factorial(n) ...
  return 0;
}
```

:::: :::: {.column width=50%} Control flow:

Call-by-value

```
int factorial(int n) {
  . . .
  return r; // function return
}
int main() {
  . . .
  // Call to factorial
  . . . factorial(n) ...
  return 0;
}
```

Call-by-value: the **value** of `n` in `main` is used to initialise the variable that (coincidentally) goes by the same name in `factorial`. The two variables are independent. Any change to `n` in `factorial` does not affect the value of `n` in main.

We will later discuss **call-by-reference**, where instead function arguments can be references to variables in the calling function.

Call-by-value (cont.)

Changes to `n` in factorial do not affect `n` in `main`, as in the following alternative implementation of `factorial`:

```
int factorial(int n) {
  int r = 1;
  while (n > 0) {
    r = r * n;
    n--; // decrement n
  }
  return r;
}
```

```
int main() {
  int n;
  . . . factorial(n) ...
  // n unchanged after the call
  return 0;
}
```

Recursion

Function calls can be recursive. This does not make any difference to a normal function call: an independent set of local variables is associated to each function call.

Recursive implementation of `factorial`:

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

Decoupled function declarations

Function declarations can appear without an associated body. This is common for instance in header files (as we will see later in the semester).

In the example, we could have:

```
// just the declaration
int factorial(int n);

int main() { . . . }

// actual implementation
int factorial(int n) { ... }
```

Conditional control flow

if and if-else instructions

if

```
if (condition)
  body
```

- body executes if condition is true (condition != 0)

if-else

```
if (condition)
  body_1
else
  body_2
```

- body_1 executes if condition is true (condition != 0), otherwise body_2 executes.

Example

Example - possible implementations of `max` to compute the maximum value of two `int` values:

- using `if`:

```
int max(int a, int b) {
  if (a > b)
    return a;
  return b;
}
```

- using `if-else`

```
int max(int a, int b) {
  if (a > b)
    return a;
  else // else is redundant
    return b;
}
```

Example (cont.)

Other variants with a single `return` point...

```
int max(int a, int b) {
  int r = b;
  if (a > b)
    r = a;
  return r;
}
```

```
int max(int a, int b) {
  int r;
  if (a > b)
    r = a;
  else
    r = b;
  return r;
}
```

Use of ternary conditional operator `? :`

More variants ... the ternary conditional operator `? :` can be employed instead:

```
int max(int a, int b) {
  int r =  a > b ? a : b;
  return r;
}
```

`r` is redundant above, as it is used just once after definition. A simpler implementation is:

```
int max(int a, int b) {
  return a > b ? a : b;
}
```

Note on syntax

A single `if`-body such as:

```
if (a > b)
  r = a;
```

is equivalent to

```
if (a > b) {
  r = a;
}
```

given that the `if` body has a single instruction.

However, **if an instruction body has more than one instruction, it must be enclosed between `{` and `}`.**

Note on syntax (cont.)

Python programmers beware:

```
if (a > b)
  r = a;
  x = y;
```

is **not equivalent** to

```
if (a > b) {
  r = a;
  x = y;
}
```

but instead equivalent to (indentation has no semantic meaning)

```
if (a > b) {
  r = a;
}
x = y;
```

Another example - leap years

U.PORTO

```
bool is_leap_year(int y) {
  ...
  // must return true IF
  // - y % 400 == 0
  // - OR ( y % 4 == 0 AND
  //        y % 100 != 0 )
}
```

A leap year has 366 days instead of 365 when its value is **(1)** divisible by 4 but not by 100,; **OR (2)** divisible by 400. For example: 2004 and 2000 are leap years, but 2005 and 2100 are not.

Simple implementation, without using `if` or `if-else`:

```
bool is_leap_year(int y) {
  return (y % 4 == 0 &&
          y % 100 != 0) ||
          y % 400 == 0;
}
```

`%`: modulo operator; `||`: logical OR; `&&`: logical AND

Leap years (cont.)

A possible implementation using `if-else`:

```
bool is_leap_year(int y) {
  bool r;
  if (y % 4 == 0 && y % 100 != 0)
    r = true;
  else if (y % 400 == 0)
    r = true;
  else
    r = false;
  return r;
}
```

Leap years (cont.)

Another alternative, this time employing nested `if` instructions:

```
bool is_leap_year(int y) {
  bool r = true;
  if (y % 4 != 0 || y % 100 == 0)
    if (y % 400 != 0)
      r = false;
  return r;
}
```

Leap years (cont.)

Another alternative, this time employing nested `if` instructions:

```
bool is_leap_year(int y) {
  bool r = true;
  if (y % 4 != 0 || y % 100 == 0)
    if (y % 400 != 0)
      r = false;
  return r;
}
```

Can you suggest another alternative?

Another example

Now consider a function to compute the number of days in a month (numbered 1 to 12):

```
int days_in_month(int m, int y) { ... }
```

The number of days is fixed for each month (30 or 31), except in the case of February, which has 29 days in a leap year and 28 in a non-leap year.

For `m` given in the range from 1 to 12:

- `days_in_month(2,y)` should return 29 when `is_leap_year(y) == true`, and 28 if not;
- `days_in_month(m,y)` should return 31 when `m` is equal to 1, 3, 5, 7, 8, 10, or 12;
- `days_in_month(m,y)` should return 30 for all other cases.

days_in_month using if-else

```
int days_in_month(int m, int y) {
  int d;
  if (m == 2)
    d = is_leap_year(y) ? 29 : 28; // February
  else if (m == 1 || m == 3 ||
           m == 5 || m == 7 ||
           m == 8 || m == 10 ||
           m == 12)
    d = 31; // Months with 31 days
  else
    d = 30; // All other months have 30 days
  return d;
}
```

days_in_month using if-else

```
int days_in_month(int m, int y) {
  int d;
  if (m == 2)
    d = is_leap_year(y) ? 29 : 28; // February
  else if (m == 1 || m == 3 ||
           m == 5 || m == 7 ||
           m == 8 || m == 10 ||
           m == 12)
    d = 31; // Months with 31 days
  else
    d = 30; // All other months have 30 days
  return d;
}
```

A **switch-case** instruction can be used instead for simpler code.

switch-case instruction

```
switch (expr) {
  case value_1:
    body_1
    break;
  case value_2:
    body_2
    break;
  . . .
  case value_n:
    body_n
    break;
  default:
    default_body
    break;
}
```

`switch-case` instruction (cont.)

Evaluates integer expression `expr`, and executes:

- `body_1` if `expr` has value `value_1`;
- `body_2` if `expr` has value `value_2`;
- ...
- `body_n` if `expr` has value `value_n`;
- `default_body` otherwise.

The `break` instruction in a `case` block makes execution proceed to the instruction following the switch-case.

days_in_month using switch-case

```
int days_in_month(int m, int y) {
  int d;
  switch (m) {
    case 2: // February
      d = is_leap_year(y) ? 29 : 28; break;
    case 1: case 3: case 5: case 7: case 8: case 10: case 12: // 31 days
      d = 31; break;
    default: // All other months have 30 days
      d = 30; break;
  }
  return d;
}
```

switch-case instruction (cont.)

What happens if we remove break for case 2: ?

```
int days_in_month(int m, int y) {
  int d;
  switch (m) {
    case 2: // February
      d = is_leap_year(y) ? 29 : 28;
      // break;
      // execution continues -->
```

```
    case 1: case 3: case 5:
    case 7: case 8: case 10:
    case 12: // Months with 31 days
      d = 31;
      break;
    . . .
  }
  return d;
}
```

switch-case instruction (cont.)

```
  switch (m) {
    case 2: // February
      d = is_leap_year(y) ? 29 : 28;
      // break; // execution continues
    case 1: case 3: case 5:
      // Months with 31 days
      . . .
      d = 31; break;
    . . .
  }
  return d;
}
```

The function will return 31 when `m` equals 2. The absence of `break` instruction for case 2 lets execution proceed to the following case block.

switch-case instruction (cont., using enum)

```
enum { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
int days_in_month(int m, int y) {
  int r;
  switch (m) {
    case FEB:
      r = is_leap_year(y) ? 29 : 28; break;
    case JAN: case MAR: case MAY: case JUL: case AUG: case OCT: case DEC:
      r = 31; break; // Months with 31 days
    default:
      r = 30; break; // All others have 30 days
  }
  return r;
}
```

Loop instructions

`while` and `do-while`

`while` loop

```
while (condition)
  body
```

- `body` executes while `condition` is true (`!= 0`).
- No iterations of `body` execute if `condition` is initially false.

`do-while` loop

```
do
  body
while (condition);
```

- Executes `body` until `condition` is `false`.
- At least one iteration of `body` executes.

Prime number check using a `while` loop

(simplest naive algorithm for prime number check)

```
bool is_prime(int n) {
  if (n <= 1)
    return false;
  int i = 2;
  while (i < n) {
    if (n % i == 0)
      return false;
    i++;
  }
  return true;
}
```

Prime number check using a `do-while` loop

U.PORTO

```
bool is_prime(int n) {
  if (n <= 1)
    return false;
  if (n == 2)
    return true;
  int i = 2;
  do {
    if (n % i == 0)
      return false;
    i++;
  } while (i < n);
  return true;
}
```

Using break in loops

A break instruction within a while and do-while loop unconditionally terminates (breaks out of) the loop. Using break can make control flow harder to understand.

```
bool is_prime(int n) {
  if (n <= 1)  return false;
  int i = 2;
  while (true) {
    if (i == n || n % i == 0)  break;
    i++;
  }
  return i == n; // prime only if i == n
}
```

Using continue in loops

A continue instruction within a while and do-while loop makes execution proceed directly to the loop condition test. Like break, continue can make control flow harder to understand.

```
bool is_prime(int n) {
  if (n <= 1)  return false;
  int i = 2;
  while (i < n) {
    if (n % i != 0) {
      i++;
      continue;
    }
    return false;
  }
```

break and continue - adequate use

Using break and continue is more appropriate for complex loops where there are several stop or re-entry conditions.

```
while (true) {
  // Multiple stop conditions
  if (stop_cond1) break;
  ...
  if (stop_cond_n) break;
  // ... re-entry conditions
  if (cont_cond_1) continue;
  ...
  if (cont_cond_k) continue;
  ...
}
```

break and continue - adequate use

Using break and continue is more appropriate for complex loops where there are several stop or re-entry conditions.

```
while (true) {
  // Multiple stop conditions
  if (stop_cond1) break;
  ...
  if (stop_cond_n) break;
  // ... re-entry conditions
  if (cont_cond_1) continue;
  ...
  if (cont_cond_k) continue;
  ...
}
```

Avoid confusing **break**, **continue**, **return**.

Avoid **return** entangled in the code.

`for` loops

`while` loops of the following form are common:

```
initialisation
while (condition) {
  body
  update
}
```

This type of iteration can be expressed more clearly using a `for` loop:

```
for (initialisation; condition; update) {
  body
}
```

Prime number check using a `for` loop

```
bool is_prime(int n) {
  if (n <= 1)
    return false;
  for (int i = 2; i < n; i++) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

An additional advantage of a `for` loop is that variables declared in the initialisation block have a scope that is limited to the loop body.

Variable `i` above can not be used outside the `for` loop.

More on functions: default arguments, function overloading

Default arguments for parameters

Functions can have **default arguments**.

We may for instance define:

```
int f(int x, int y=2, int z=3) {
  return x + y + z;
}
```

Function `f` can then be called omitting values for `z` or both `y` and `z`.

```
int v = f(1)          // equivalent to f(1, 2, 3)
      + f(1, 0)       // equivalent to f(1, 0, 3)
      + f(1, 0, 0);
```

Only trailing parameters can have default arguments. We cannot for instance define:

```
int g(int x=1, int y) { . . . }
// ERROR
  error: missing default argument
  on parameter 'x'
```

→ Further reference

Function overloading

Function overloading: we can have several definitions of functions with the same name, called function overloads, as long as they have different parameters (in number or type).

For instance, we can have several declarations of a function `f` as follows:

```
int f(int x);
int f(double x);
int f(int x, int y);
void f();
double f(double x, double y);
```

In contrast, the following definitions are invalid for g, since both function overloads have the same arguments:

```
int g(int x);
void g(int x);

error: functions that differ
only in their return type cannot
be overloaded
```

Function overloading (cont.)

Ambiguities may arise at function calls. The following overloads are valid:

```
int h(double x=1.2);
int h(double x, int y=2, int z=3);
int h(float x);
```

Some calls may be ambiguous:

```
h();        // OK => 1st overload h(1.2)
h(1.2, 5); // OK => 2nd overload h(1.2, 5, 3)
h(1.2f);    // OK => 3rd overload
h(1.0);     // ERROR -> 1st or 2nd overload?
```

```
  error: call to 'h' is ambiguous
```

Function overloading (cont.)

```
int h(double x=1.2);
int h(double x, int y=2, int z=3);
int h(float x);
```

Another example of an ambiguous call:

```
// There is no overload with int parameter
// but int expressions can be promoted
// to float or double type
int v = 123;
h(v); // ERROR - 1st, 2nd, or 3rd overload?
```

```
error: call to 'h' is ambiguous
```

References, call-by-reference

References

In C++ we can declare a variable to be a **reference**, when its type has the form `type&`. For instance, in

```
int  v = 1;
int& r = v;
```

the `int&` type for `r` indicates that `r` is a reference variable `v` of type `int`.

What are the implications? `r` refers to `v`, hence any updates using `r` are reflected on `v` and vice-versa.

```
std::cout << v << ' '
     << r << '\n';
     // --> 1 1
```

```
v++;
std::cout << v << ' '
     << r << '\n';
     // --> 2 2
```

```
r++;
std::cout << v << ' '
     << r << '\n';
     // --> 3 3
```

References (cont.)

References are initialized **once** in their declaration.

```
int v1 = 10;
int v2 = 20;
// initialisation of r
int& r = v1;
// equivalent to v1 = v2
r = v2;
// increment of v2 does not change v1
v2++;
cout << r << ' '
     << v1 << ' '
     << v2 << '\n'; // --> 20 20 21
```

There can be more than one reference to a variable.

```
int  v  = 0;
int& r1 = v;  // r1 refers to v
int& r2 = v;  // r2 refers to v
int& r3 = r2; // r3 refers to v
              // (reference copy)
r1++; r2++; r3++;
std::cout << v << '\n'; // --> ?
```

What does it return?

References (cont.)

References are initialized **once** in their declaration.

```
int v1 = 10;
int v2 = 20;
// initialisation of r
int& r = v1;
// equivalent to v1 = v2
r = v2;
// increment of v2 does not change v1
v2++;
cout << r << ' '
     << v1 << ' '
     << v2 << '\n'; // --> 20 20 21
```

There can be more than one reference to a variable.

```
int  v  = 0;
int& r1 = v;  // r1 refers to v
int& r2 = v;  // r2 refers to v
int& r3 = r2; // r3 refers to v
              // (reference copy)
r1++; r2++; r3++;
std::cout << v << '\n'; // --> ?
```

What does it return?

--> 3

Call-by-reference

The use of references enables **call-by-reference** semantics.

Consider the following function:

```
void get_min_max(int a, int b, int& min, int& max) {
  min = a < b ? a : b;
  max = a > b ? a : b;
}
```

In `get_min_max`:

- `a` and `b` are call-by-value parameters;
- `min` and `max` are **call-by-reference** parameters.

What does this mean?

Call-by-reference (cont.)

```
void get_min_max(int a, int b,
                 int& min, int& max) {
  min = a < b ? a : b;
  max = a > b ? a : b;
}
int main() {
  int x = 200, y = 100;
  int m, M;
  get_min_max(x, y, m, M);
  // ^-- call to get_min_max
  std::cout << m < ' ' << M;
  return 0;
}
```

In the call to `get_min_max` by `main`:

- `x` and `y` provide the values for parameters `a` and `b`
- `m` and `M` provide references for parameters `min` and `max`.

There is no need to declare reference variables in `main`: references are created implicitly by the function call.

Call-by-reference (cont.)

```
void get_min_max(int a, int b,
                 int& min, int& max) {
  min = a < b ? a : b;
  max = a > b ? a : b;
}
int main() {
  int x = 200, y = 100;
  int m, M;
  get_min_max(x, y, m, M);
  // ^-- call to get_min_max
  std::cout << m < ' ' << M; // <-- m=100, M=200
  return 0;
}
```

Assignments to `min` and `max` within `get_min_max` will write variables `m` and `M`.

After `get_min_max` returns, `m` will contain value 100 and `M` will contain value 200.

Call-by-reference (cont.)

Does the following variant of `get_min_max` **work as expected**?

```
void get_min_max(int a, int b, int& min, int& max) {
  // Swap values of a and b if necessary s.t. a <= b
  if (a > b) { int tmp = a; a = b; b = tmp; }
  min = a; max = b;
}
int main() {
  int x = 200, y = 100;
  ...
  get_min_max(x, y, m, M); // <-- call to get_min_max
  // x and y SHOULD BE UNCHANGED after the call
  ...
}
```

Call-by-reference (cont.)

Does the following variant of `get_min_max` **work as expected**?

```
void get_min_max(int a, int b, int& min, int& max) {
  // Swap values of a and b if necessary s.t. a <= b
  if (a > b) { int tmp = a; a = b; b = tmp; }
  min = a; max = b;
}
int main() {
  int x = 200, y = 100;
  ...
  get_min_max(x, y, m, M); // <-- call to get_min_max
  // x and y SHOULD BE UNCHANGED after the call
  ...
}
```

YES! As discussed earlier, changing the call-by-value parameters `a` and `b` within `get_min_max` does not affect variables `x` and `y` in the caller function.