Fundamentals of Programming I

4 Procedural Abstraction

Grado en Ingeniería Informática

Luis Hernández Yáñez
Facultad de Informática
Universidad Complutense

Index

Top-Down Design: Tasks and Subtasks

Luis Hernández Yáñez

Tasks and Subtasks

Successive Refinements

Tasks a program has to execute: Can be divided into easier subtasks

Subtasks: Also can be divided into other, easier subtasks...

→ Successive refinements

Designing in successive steps that expand the level of detail

Examples:

- ✓ Draw 
- ✓ Print the string HELLO MUM in big letters

Luis Hernández Yáñez

A Drawing

1. Draw 
2. Draw 
3. Draw 

REFINEMENT

1. Draw 
2. Draw 
   - 2.1. Draw 
   - 2.2. Draw
3. Draw 

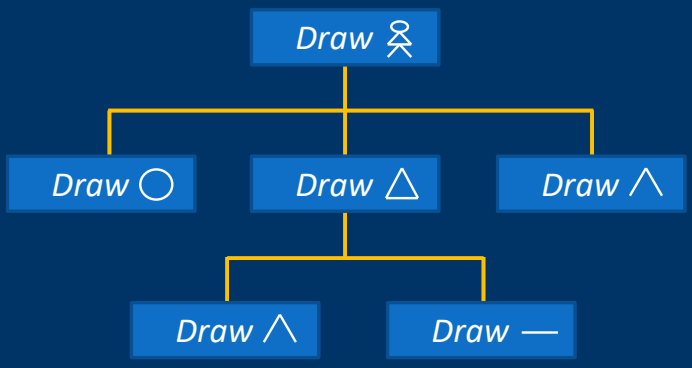
Same task

A Drawing

1. Draw 
2. Draw 
   - 2.1. Draw 
   - 2.2. Draw
3. Draw 

4 tasks, but two of them are the same
We only need to know how to draw:

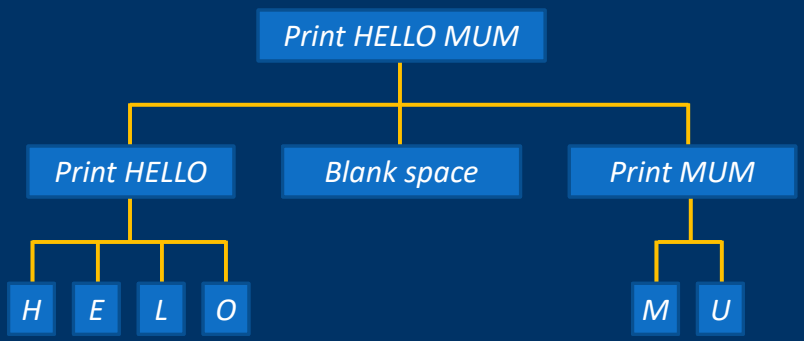
A Drawing




```
void drawCircle()
{ ... }

void drawSecants()
{ ... }

void drawLine()
{ ... }

void drawTriangle()
{
   drawSecants();
   drawLine();
}

int main() {
   drawCircle();
   drawTriangle();
   drawSecants();
   return 0;
}
```


Message in Big Letters

Print the string HELLO MUM in big letters



Basic tasks

Message in Big Letters

```
void printH() {
   cout << "*   *" << endl;
   cout << "*   *" << endl;
   cout << "*****" << endl;
   cout << "*   *" << endl;
   cout << "*   *" << endl << endl;
}

void printE() {
   cout << "*****" << endl;
   cout << "*    " << endl;
   cout << "***  " << endl;
   cout << "*    " << endl;
   cout << "*****" << endl << endl;
}

void printL()
{ ... }

void printO()
{ ...}
```

```
void blankSpace() {
   cout << endl << endl << endl;
}

void printM()
{ ...}

void printU()
{ ...}

int main() {
   printH();
   printE();
   printL();
   printL();
   printO();
   blankSpace();
   printM();
   printU();
   printM();
   return 0;
}
```

Luis Hernández Yáñez

Fundamentals of Programming I

Subprograms

Luis Hernández Yáñez

Procedural Abstraction

Subprograms

Smaller programs inside other programs

- ✓ Independent execution units
- ✓ Encapsulate code and data
- ✓ Can communicate with other subprograms (data exchange)

Higher abstraction level for the program
Easier testing, debugging and maintenance

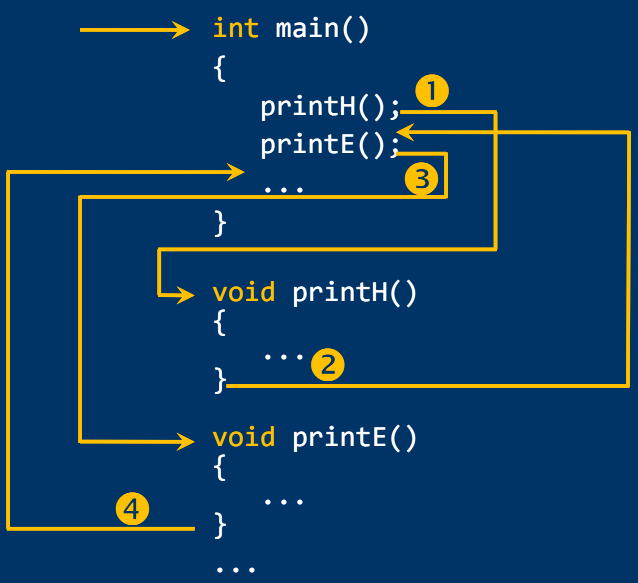
Subroutines, procedures, functions, actions, ...

- ✓ Implement individual tasks in the program
- ✓ Concrete, identifiable and coherent functionality (design)
- ✓ Execute from beginning to end when called (*invoked*)
- ✓ End returning control to calling point

Luis Hernández Yáñez

Subprograms

Execution flow



Luis Hernández Yáñez

Subprograms

Subprograms in C++

General form of a C++ subprogram:

```
type name(parameters) // Heading
{
    // Body
}
```

- ✓ *Type* of data the subprogram returns as a result
- ✓ *Parameters* for communicating with the exterior
- ✓ *Body*: A block of code!

Luis Hernández Yáñez

Subprograms

Kinds of subprograms

Procedures (*actions*):

- DON'T `return` any data as subprogram result
- Type: `void`
- Calling: Independent instruction `mostrarH();`

Functions:

- DO `return` a subprogram result
- Type different than `void`
- Calling: Inside any valid expression `x = 12 * + square(20) - 3;`
- Call is substituted in the expression by the value returned
- *We are using functions since Lesson 2!*

Luis Hernández Yáñez

Subprograms

Functions

Subprograms of type different than `void`

```
...
int menu()
{
   int op;
   cout << "1 - Edit" << endl;
   cout << "2 - Combine" << endl;
   cout << "3 - Publish" << endl;
   cout << "0 - Cancel" << endl;
   cout << "Option: ";
   cin >> op;
   return op;
}
```

```
int main()
{
   ...
   int option;
   option = menu();
   ...
```

Subprograms

Procedures

Subprograms of type `void`

```
...
void menu() {
   int op;
   cout << "1 - Edit" << endl;
   cout << "2 - Combine" << endl;
   cout << "0 - Cancel" << endl;
   cout << "Option: ";
   cin >> op;
   if (op == 1)
      edit();
   else if (op == 2)
      combine();
}
```

```
int main()
{
   ...
   menu();
   ...
```

Subprograms and Data

Luis Hernández Yáñez

Subprograms and Data

Subprogram's exclusive use

```
type name(parameters) // Heading
{
   Local declarations // Body
}
```

- ✓ Local declarations of types, constants and variables
  Inside subprogram's body
- ✓ Parameters declared in subprogram's heading
  Communication with other subprograms

Luis Hernández Yáñez

Local and Global Data

Program data

- ✓ Global data: declared outside every subprogram
  Exists during entire program execution
- ✓ Local data: declared in any subprogram.
  Exists only during subprogram execution

Scope and visibility of data *Lesson 3*

- Scope of global data: rest of the program
  Known inside following subprograms
- Scope of local data: rest of subprogram
  Not known outside subprogram
- Data visibility: Local data in a block hides external data with same name

Local and Global Data

```
#include <iostream>
using namespace std;

const int MAX = 100;
double income;

...
void proc() {
   int op;
   double income;
   ...
}

int main() {
   int op;
   ...
   return 0;
}
```

Global Data

Data local to `proc()`

Known: `MAX` (global), `op` (local) and `income` (local that hides global)

Data local to `main()`

Known: `MAX` (global), `op` (local) and `income` (global)

`op` in `proc()` is different from `op` in `main()`

Local and Global Data

About using global data in programs

Global data MUST NOT BE USED in subprograms

✓ *Need external data?*

Define parameters in subprogram

External data will be passed as arguments in subprogram's call

✓ Use of global data in subprograms:

Risk of *lateral effects*

Unnoticed modification of the data, affecting other places

Exceptions:

✓ Global constants (unalterable values)

✓ Global types (needed in several subprograms) (no data)

Luis Hernández Yáñez

Fundamentals of Programming I

Parameters

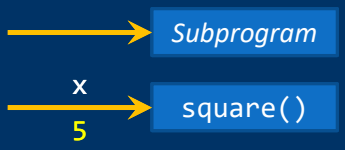
Luis Hernández Yáñez

Communication with other Subprograms

Input data, output data and input/output data

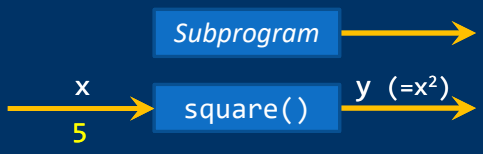
Input Data: Accepted

Subprogram that given a number prints its square on the screen:



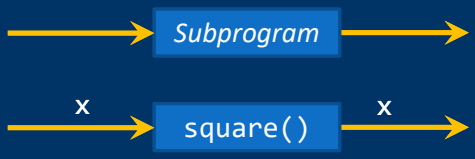
Output Data: Returned

Subprogram that given a number returns its square:



Input/Output Data: Accepted and updated

Subprogram that given a numerical variable returns it squared:



Luis Hernández Yáñez

Parameters in C++

Parameter declaration

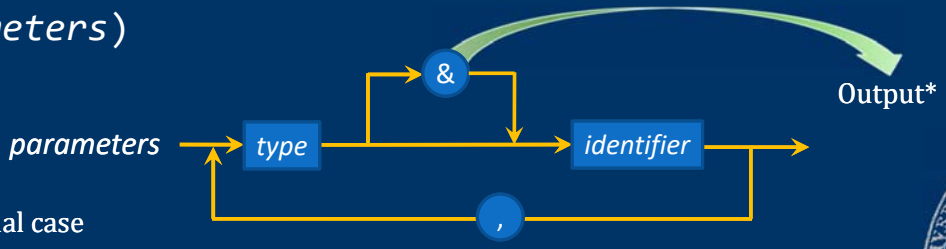
Only two kinds of parameters in C++:

- Input only (*by value*)
- Output (only output or I/O) (*by reference / by variable*)

Formal parameter list

Between the parentheses in subprogram's heading

```
type name(parameters)
```



(*) Arrays are a especial case

Luis Hernández Yáñez

Parameters Passed by Value

Receive copies of the arguments used in subprogram's call

```
int square(int num)
double power(double base, int exp)
void print(string name, int age, string nif)
void proc(char c, int x, double a, bool b)
```

Receive their values when subprogram is called

Arguments: Expressions in general

Variables, constants, literals, function calls, operations

Destroyed when subprogram execution ends

Warning! Arrays are passed by value as constants:

```
double mean(const tArray list)
```

Parameters Passed by Reference &

Same identity as variables passed as arguments

```
void increment(int &x)
void interchange(double &x, double &y)
void proc(char &c, int &x, double &a, bool &b)
```

Receive the variable used in subprogram call: *Only variables!*

Arguments can be modified

We will not use parameters passed by reference in functions!

Only in procedures

There may be some passed by value and some by reference

Warning! Arrays are passed by reference without using &:

```
void insert(tArray list, int &counter, double item)
```

The argument of `list` (variable `tArray`) will be modified

Arguments

Luis Hernández Yáñez

Calling Subprograms with Parameters

name(*arguments*)

- As many arguments as parameters, and in the same order
- Type agreement between argument and parameter
- By value: Valid expressions (the result is passed)
- By reference: *Only variables!*

Values of expressions passed by value are copied in corresponding parameters

Arguments passed by reference (variables) are bound to corresponding parameters

Luis Hernández Yáñez

Arguments Passed by Value

Valid expressions with type agreement:

```
void proc(int x, double a)
```

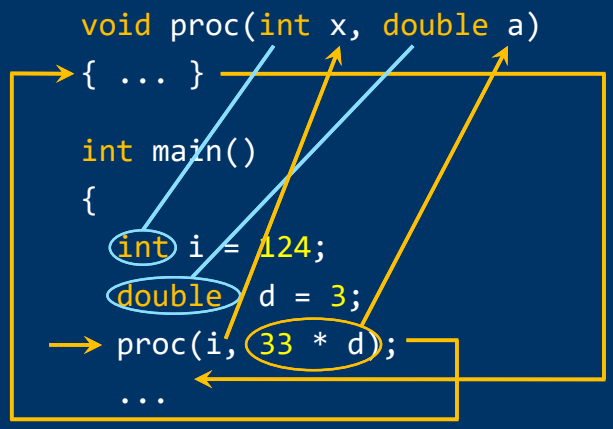
→ `proc(23 * 4 / 7, 13.5);`

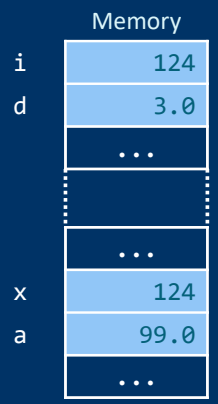
→
```
double d = 3;
proc(12, d);
```

→
```
double d = 3;
int i = 124;
proc(i, 33 * d);
```

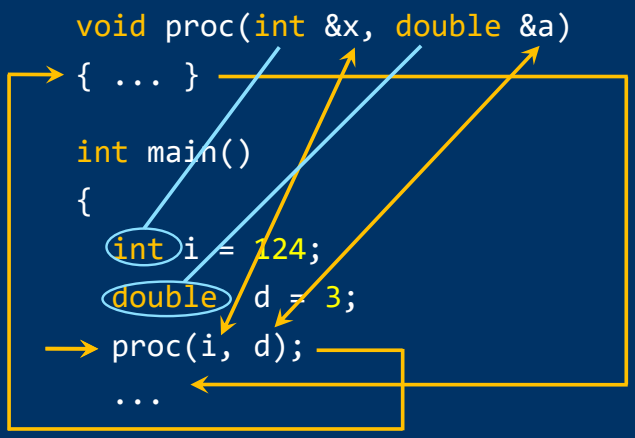
→
```
double d = 3;
int i = 124;
proc(sqr(20) * 34 + i, i * d);
```

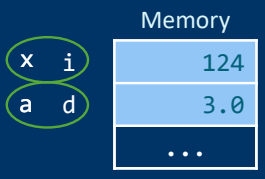

Arguments Passed by Value





Arguments Passed by Reference





Which Calls are Correct?

With these declarations:

```
int i;
double d;
void proc(int x, double &a);
```

Are the following arguments passing correct? Why not?

```
proc(3, i, d);
proc(i, d);
proc(3 * i + 12, d);
proc(i, 23);
proc(d, i);
proc(3.5, d);
proc(i);
```

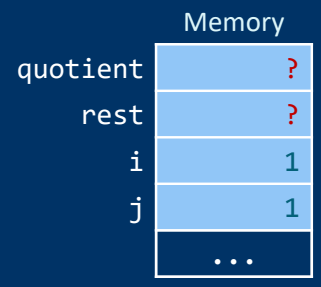
✘ Nr. of arguments ≠ Nr. of parameters
✓
✓
✘ Parameter by reference → Variable!
✘ `double` argument for `int` parameter!
✘ `double` argument for `int` parameter!
✘ Nr. of arguments ≠ Nr. of parameters

Argument Passing

```
void divide(int op1, int op2, int &div, int &rem) {
// Divides op1 by op2 and returns quotient and reminder
   div = op1 / op2;
   rem = op1 % op2;
}

int main() {
   int quotient, rest;
   for (int j = 1; j <= 4; j++)
      for (int i = 1; i <= 4; i++) {
         divide(i, j, quotient, rest);
         cout << i << " by " << j << " gives a quotient of "
            << quotient << " and a remainder of " << rest << endl;
      }

   return 0;
}
```

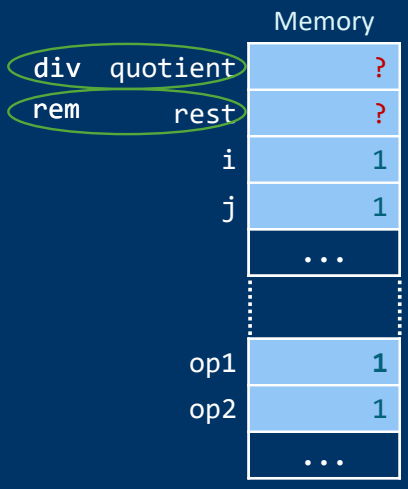

Argument Passing

```
void divide(int op1, int op2, int &div, int &rem) {
// Divides op1 by op2 and returns quotient and reminder
   div = op1 / op2;
   rem = op1 % op2;
}

int main() {
   int quotient, rest;
   for (int j = 1; j <= 4; j++)
      for (int i = 1; i <= 4; i++) {
         divide(i, j, quotient, rest);
         ...
      }

   return 0;
}
```

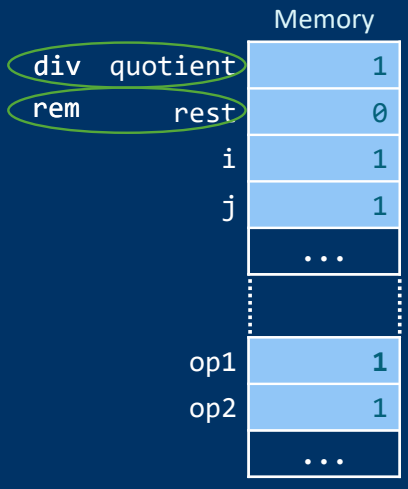
Argument Passing

```
void divide(int op1, int op2, int &div, int &rem) {
// Divides op1 by op2 and returns quotient and reminder
   div = op1 / op2;
   rem = op1 % op2;
}

int main() {
   int quotient, rest;
   for (int j = 1; j <= 4; j++)
      for (int i = 1; i <= 4; i++) {
         divide(i, j, quotient, rest);
         ...
      }

   return 0;
}
```



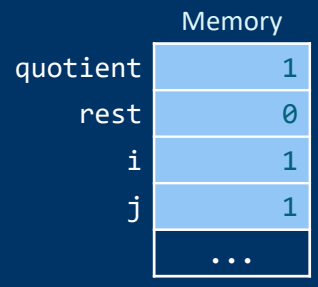
Argument Passing

```
void divide(int op1, int op2, int &div, int &rem) {
// Divides op1 by op2 and returns quotient and reminder
   div = op1 / op2;
   rem = op1 % op2;
}

int main() {
   int quotient, rest;
   for (int j = 1; j <= 4; j++)
      for (int i = 1; i <= 4; i++) {
         divide(i, j, quotient, rest);
         ...
      }

   return 0;
}
```



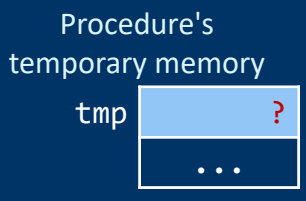
Argument Passing

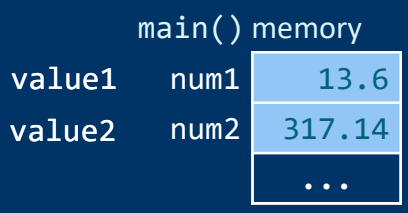
```
void divide(int op1, int op2, int &div, int &rem) {
// Divides op1 by op2 and returns quotient and reminder
   div = op1 / op2;
   rem = op1 % op2;
}

int main() {
   int quotient, rest;
   for (int j = 1; j <= 4; j++)
      for (int i = 1; i <= 4; i++) {
         divide(i, j, quotient, rest);
         ...
      }

   return 0;
}
```



More Examples

```
void exchange(double &value1, double &value2) {
// Exchanges the values
   double tmp; // Local variable (temporary)
   tmp = value1;
   value1 = value2;
   value2 = tmp;
}

int main() {
   double num1, num2;
   cout << "Value 1: ";
   cin >> num1;
   cout << "Value 2: ";
   cin >> num2;
   exchange(num1, num2);
   ...
   return 0;
}
```





More Examples

```
// Prototype
void change(double price, double payment, int &euros, int &cent50,
   int &cent20, int &cent10, int &cent5, int &cent2, int &cent1);

int main() {
   double price, payment;
   int euros, cent50, cent20, cent10, cent5, cent2, cent1;
   cout << "Price: ";
   cin >> price;
   cout << "Payment: ";
   cin >> payment;
   change(price, payment, euros, cent50, cent20, cent10, cent5, cent2, cent1);
   cout << "Change: " << euros << " euros, " << cent50 << " x 50c., "
        << cent20 << " x 20c., " << cent10 << " x 10c., "
        << cent5 << " x 5c., " << cent2 << " x 2c. and "
        << cent1 << " x 1c." << endl;
   return 0;
}
```


More Examples

```
void change(double price, double payment, int &euros, int &cent50,
            int &cent20, int &cent10, int &cent5, int &cent2, int &cent1) {
   if (payment < price) // Insufficient payment
      cout << "Error: Payment less than price!" << endl;
   else {
      int chng = int(100.0 * (payment - price) + 0.5);
      euros  = chng / 100;
      chng = chng % 100;
      cent50 = chng / 50;
      chng = chng % 50;
      cent20 = chng / 20;
      chng = chng % 20;
      cent10 = chng / 10;
      chng = chng % 10;
      cent5  = chng / 5;
      chng = chng % 5;
      cent2  = chng / 2;
      cent1  = chng % 2;
   }
}
```


Notifying Errors

We can detect errors during subprogram execution

Errors that impede making some computations:

```
void change(double price, double payment, int &euros, int &cent50,
   int &cent20, int &cent10, int &cent5, int &cent2, int &cent1) {
   if (payment < price) // Insufficient payment
      cout << "Error: Payment is less than price!" << endl;
   ...
```

Should the subprogram notify the user or notify the program?

→ It's better to notify the calling point and decide what to do there

```
void change(double price, double payment, int &euros, int &cent50,
   int &cent20, int &cent10, int &cent5, int &cent2, int &cent1,
   bool &error) {
   if (payment < price) // Insufficient payment
      error = true;
   else {
      error = false;
   ...
```


Notifying Errors

change.cpp

At the calling point we decide what to do in case of error...

- ✓ Inform the user?
- ✓ Ask for data again?
- ✓ Etcetera

```
int main() {
   double price, payment;
   int euros, cent50, cent20, cent10, cent5, cent2, cent1;
   bool error;
   cout << "Price: ";
   cin >> price;
   cout << "Payment: ";
   cin >> payment;
   change(price, payment, euros, cent50, cent20, cent10, cent5, cent2,
          cent1, error);
   if (error)
      cout << "Error: Payment is less than price!" << endl;
   else {
      ...
```


Function Result

Luis Hernández Yáñez

Function Result

A function must return a result

The function ends its execution by returning a result

`return` instruction:

- Returns the data that follows as the function result
- Ends function execution

The data returned substitutes the function call in the expression

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

This instruction will never be executed

```
int main() {
    cout << 2 * square(16);

    return 0;
}
```

256

Luis Hernández Yáñez

Example: Factorial

factorial.cpp

Factorial (N) = 1 x 2 x 3 x ... x (N-2) x (N-1) x N

```
long long int factorial(int n); // Prototype

int main() {
   int num;
   cout << "Num: ";
   cin >> num;
   cout << "Factorial of " << num << ": " << factorial(num) << endl;
   ...

long long int factorial(int n) {
   long long int fact = 1;
   if (n < 0)
      fact = 0;
   else
      for (int i = 1; i <= n; i++)
         fact = fact * i;
   return fact;
}
```


Only One Exit Point

```
int compare(int val1, int val2) {
// -1 if val1 < val2, 0 if equal, +1 if val1 > val2
   if (val1 == val2) {
      return 0;
   }
   else if (val1 < val2) {
      return -1;
   }
   else {
      return 1;
   }
}
```

3 exit points!

Only One Exit Point

```
int compare(int val1, int val2) {
// -1 if val1 < val2, 0 if equal, +1 if val1 > val2
   int result;

   if (val1 == val2) {
      result = 0;
   }
   else if (val1 < val2) {
      result = -1;
   }
   else {
      result = 1;
   }

   return result;
}
```

→ Only one exit point ✓

Luis Hernández Yáñez

Subprogram Finalization

Procedures (type `void`):

— When reaching the close brace that ends the subprogram or

— After executing a `return` instruction (without result)

Functions (type different than `void`):

— ONLY after executing a `return` instruction (with a result)

Our subprograms will always finish at the end:

✓ We will not use `return` in procedures

✓ Functions: only one `return` at the end

For easier debugging and maintenance code subprograms with a unique exit point

Luis Hernández Yáñez

Prototypes

Luis Hernández Yáñez

Subprograms of the Program

Where do we put them? Before `main()`? After `main()`?

→ We will put them after `main()`

Are subprogram callings correct?

In `main()` or in other subprograms

- Does the subprogram exist?
- Is there agreement between arguments and parameters?

There must be, before `main()`, prototypes for all subprograms

Prototype: subprogram heading ended with `;`

```
void drawCircle();
void printM();
void proc(double &a);
int square(int x);
...
```

`main()` is the only subprogram that doesn't need to be prototyped

Luis Hernández Yáñez

BY NC SA

Examples

exchange.cpp

```
#include <iostream>
using namespace std;

void exchange(double &value1, double &value2); // Prototype

int main() {
   double num1, num2;
   cout << "Value 1: ";
   cin >> num1;
   cout << "Value 2: ";
   cin >> num2;
   exchange(num1, num2);
   ...

void exchange(double &value1, double &value2) {
   double tmp; // Local variable (temporary)
   tmp = value1;
   value1 = value2;
   value2 = tmp;
}
```

Make sure that prototypes match implementations

Examples

math.cpp

```
#include <iostream>
using namespace std;

// Prototypes
long long int factorial(int n);
int summation(int n);

int main() {
   int num;
   cout << "Num: ";
   cin >> num;
   cout << "Factorial of "
        << num << ": "
        << factorial(num) << endl
        << "Summation from 1 to "
        << num << ": "
        << summation(num) << endl;

   return 0;
}
```

```
long long int factorial(int n) {
   long long int fact = 1;

   if (n < 0)
      fact = 0;
   else
      for (int i = 1; i <= n; i++)
         fact = fact * i;

   return fact;
}

int summation(int n) {
   int sum = 0;

   for (int i = 1; i <= n; i++)
         sum = sum + i;

   return sum;
}
```


Operator Functions

Luis Hernández Yáñez

Infix Notation (operator)

leftOperand operator rightOperand

```
a + b
```

The operator is executed with both operands as arguments

Operators are implemented as functions:

```
type operatorsymbol(parameters)
```

For unary operators there will be only one parameter

For binary operators there will be two parameters

The *symbol* is an operator symbol (one or two characters):

+, -, *, /, --, <<, %, ...

Luis Hernández Yáñez

Operator Functions

```
double prod(tVector v1, tVector v2);
                    tVector a, b;
                    cout << prod(a, b);

double operator*(tVector v1, tVector v2);
                    tVector a, b;
                    cout << a * b;
```

Implementation will be exactly the same!

Closer to mathematical language

Fundamentals of Programming I

Top-Down Design (Example)

Successive Refinements

Initial specification (Step 0).-

Develop a program to make conversion operations for measures until the user decides to quit

Analysis and design increasing the level of detail in every step

What conversion operations?

Step 1.-

Develop a program to make conversion operations for measures until the user decides to quit

* *Inches to centimeters*
* *Pounds to grams*
* *Degrees Fahrenheit to degrees Celsius*
* *Gallons to liters*

Luis Hernández Yáñez

Successive Refinements

Step 2.-

Develop a program that shows a menu with four measure conversion operations:

* *Inches to centimeters*
* *Pounds to grams*
* *Degrees Fahrenheit to degrees Celsius*
* *Gallons to liters*

And then reads user's choice and proceeds with the conversion until the user decides to quit

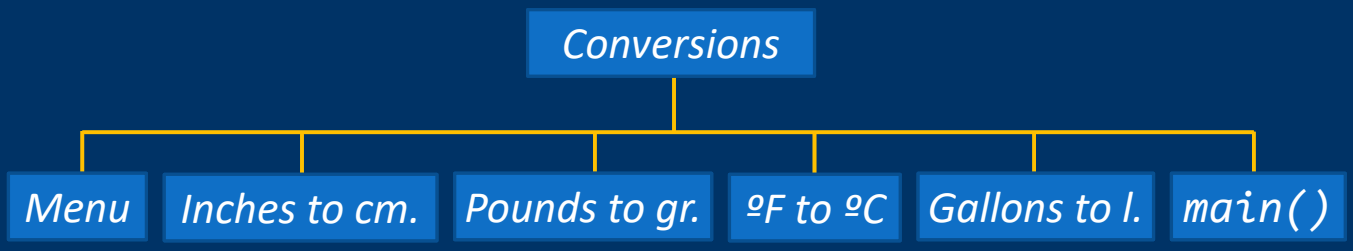
6 general tasks:

Menu, four conversion functions and `main()`

Luis Hernández Yáñez

Successive Refinements

Step 2.-



Successive Refinements

Step 3.-

* *Menu:* Show four options plus one for quitting
  Ask the user for an option, validate it and return it if valid
* *Inches to centimeters:*
  Return the equivalent in centimeters for the value in inches
* *Pounds to grams:*
  Return the equivalent in grams for the value in pounds
* *Degrees Fahrenheit to degrees Celsius:*
  Return the equivalent in ºC for the value in ºF
* *Gallons to liters:*
  Return the equivalent in liters for the value in gallons
* *Main program:* `main()`

Successive Refinements

Step 3.- Each task becomes a subprogram

Communication between subprograms:

Function	Input	Output	Returned value
menu()	–	–	int
inchToCm()	double inches	–	double
pdToGr()	double pounds	–	double
fahrToCel()	double degrees	–	double
galToLit()	double gallons	–	double
main()	–	–	int

Successive Refinements

Step 4.- Detailed algorithms for each subprogram → Programming

```
#include <iostream>
using namespace std;
// Prototypes
int menu();
double inchToCm(double inches);
double pdToGr(double pounds);
double fahrToCel(double degrees);
double galToLit(double gallons);

int main() {
   double value;
   int op = -1;
   while (op != 0) {
      op = menu();
      ...
   }
   return 0;
}
```

.../...

Successive Refinements

```
        switch (op) {
        case 1:
           cout << "Inches: ";
           cin >> value;
           cout << inchToCm(value) << " cm." << endl;
           break;
        case 2:
           cout << "Pounds: ";
           cin >> value;
           cout << pdToGr(value) << " gr." << endl;
           break;
        case 3:
           cout << "Degrees Fahrenheit: ";
           cin >> value;
           cout << fahrToCel(value) << " ºC" << endl;
           break;
        case 4:
           cout << "Gallons: ";
           cin >> value;
           cout << galToLit(value) << " l." << endl;
           break;
        }
```

.../...

Luis Hernández Yáñez

Successive Refinements

```
int menu() {
   int op = -1;

   while ((op < 0) || (op > 4)) {
      cout << "1 - Inches to Cm." << endl;
      cout << "2 - Pounds to Gr." << endl;
      cout << "3 - Fahrenheit to ºC" << endl;
      cout << "4 - Gallons a L." << endl;
      cout << "0 - Quit" << endl;
      cout << "Option: ";
      cin >> op;
      if ((op < 0) || (op > 4))
         cout << "Invalid option!" << endl;
   }

   return op;
}
```

.../...

Luis Hernández Yáñez

Successive Refinements

conversions.cpp

```
double inchToCm(double inches) {
   const double cmPerInch = 2.54;
   return inches * cmPerInch;
}

double pdToGr(double pounds) {
   const double grPerPnd = 453.6;
   return pounds * grPerPnd;
}

double fahrToCel(double degrees) {
   return ((degrees - 32) * 5 / 9);
}

double galToLit(double gallons) {
   const double ltrPerGal = 4.54609;
   return gallons * ltrPerGal;
}
```

Luis Hernández Yáñez

Fundamentals of Programming I

Preconditions and postconditions

Luis Hernández Yáñez

Preconditions and postconditions

Subprogram integrity

Conditions that must be satisfied before execution

→ Preconditions

✓ When calling the subprogram they must be guaranteed

Conditions that will be satisfied after finishing execution

→ Postconditions

✓ When returning to the calling point they are guaranteed

Assertions:

Conditions that if not true execution is halted

Function `assert()`

Assertions as Preconditions

Preconditions

For example, we will not convert negative measures:

```
double inchToCm(double inches) {
   assert(inches > 0);

   double cmPerInch = 2.54;

   return inches * cmPerInch;
}
```

The function has a precondition: `inches` must be positive

`assert(inches > 0);` will interrupt execution if not true

Assertions as Preconditions

Preconditions

When calling the subprogram they must be guaranteed:

```
int main() {
   double value;
   int op = -1;
   while (op != 0) {
      op = menu();
      switch (op) {
      case 1:
            cout << "Inches: ";
            cin >> value;
            if (value < 0)
               cout << "Invalid!" << endl;
            else { // Precondition is guaranteed...
               ...
```

Luis Hernández Yáñez

Assertions as Postconditions

Postconditions

A subprogram can guarantee conditions at the end:

```
int menu() {
   int op = -1;
   while ((op < 0) || (op > 4)) {
      ...
      cout << "Option: ";
      cin >> op;
      if ((op < 0) || (op > 4))
         cout << "Invalid option!" << endl;
   }
   assert ((op >= 0) && (op <= 4));
   return op;
}
```

The subprogram should make sure they are satisfied

Luis Hernández Yáñez

Promote Open Culture!

Creative Commons License

Attribution
You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Non commercial
You may not use this work for commercial purposes.

Share alike
If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

Click on the upper right image to learn more...

Luis Hernández Yáñez