DIGITAL ARTS & ENTERTAINMENT

Functions

Part II: getting serious

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

Contents

- default parameters
- function reference parameters
- return by reference
- recursive functions

Default parameters

- C++ allows us to write two or more overloaded functions as one function, as follows:

```
void DrawHouse(int x = 10, int y = 30);
```

- This means: “if no values are given for x and y, use the values of 10 and 30”.
- If values are given, they have priority over the default values.

Default parameters

- C++ allows us to write two or more overloaded functions as one function, as follows:

```
void DrawHouse(int x = 10, int y = 30);
```

- Calling this function:

```
DrawHouse(); // x=10, y=30
DrawHouse(15); // x=15, y=30
DrawHouse(20,15); // x=20, y=15
```

Default parameters : Example

```
// prototype
int Multiply(int a, int b = 10);
// usage
int main()
{
  int result{};
  result = Multiply(10);
  std::cout << result << '\n'; // 100

  result = Multiply(10, 5);
  std::cout << result << '\n'; // 50
}
// definition
int Multiply(int a, int b)
{
  return a * b;
}
```

Contents

- default parameters
- Function reference parameters
- return by reference
- recursive functions

Pass by value

```
int main()
{
	int number { };
	CallByValue(number);
	std::cout << number << endl;  // 0 will be printed!
}
void CallByValue(int a)
{
	a = a + 2;
}
```

a is a copy of number!!

the copy is incremented

Pass by value

- Advantages:
  - Security: Arguments are never changed by the function being called.
- Disadvantages:
  - Copying big variables can incur a significant performance penalty.
    - string, structs, classes, arrays (see later)
  - Copying can be avoided by passing by reference.

Reference types

- A reference variable type is an alias to an existing variable.
- It's like "A second name for the variable".
- Declaration + initialization: add a & after the type
  - `int& ref;` -> Error! Must be initialized
  - `int& ref{};` -> Error! Must be initialized using a variable.
  - `int& ref{14};` -> Error! A literal is not a variable.
  - `int a{2}`
  - `int& ref{a};` > Finaly ok! *ref* is a reference to *a*

Reference types

```
int a{2};
int& ref{a};
a = 5;
std::cout << ref; // what is printed ?
```

5

Pass by reference

```
int main()
{
	int number{ 0 };
	CallByReference(number);
	std::cout << number << '\n';
}
void CallByReference(int& a)
{
	a = a + 2;
}
```

a is a reference !!

A reference is an alias of the original variable. Is like having a variable with two names.

Changing the value of a reference is changing the original.

number is 2

Call by reference→When to use?

1. For basic types: only when the value of the passed variable needs to be changed by the function.
2. For other types: always, to avoid large copy operations (std::string, Point2f, etc…).

Call by reference→ Pro / Con

- Disadvantage
  - Values can be modified in the function.
- Advantage
  - Values can be modified in the function
  - Avoids the copy operation

What if you want to avoid the copy but do not want to modify the arguments?

->Pass by const reference

Pass by const reference (const modifier)

- To prevent that the function modifies the referenced variables.
  - Make the reference “read-only”.
  - `void Print(const std::string& s);`
- Compiler error when there is an attempt to modify.
- Always use const ref when passing larger objects such as strings or functions, that do not need modification inside the function being called.

DEMO: Three functions:

- Pass by value

```
void DrawString1(std::string text);
```

- Pass by reference

```
void DrawString2(std::string &text);
```

- Pass by const reference

```
void DrawString3(const std::string &text);
```

Pass by value

```
int main()
{
  std::string s = "I love programming!\n";
  DrawString1(s);
  std::cout << s;
}
void DrawString1(std::string text)
{
  std::cout << text;
  // we try to be evil -> FAILING:
  text = "I really hate it!\n";
}
```

Pass by reference

```
int main()
{
  std::string s = "I love programming!\n";
  DrawString2(s);
  std::cout << s;
}
void DrawString2(std::string& text)
{
  std::cout << text;
  // we try to be evil -> SUCCESS:
  text = "I change the original string through the reference.\n";
}
```

Pass by const reference

```
int main()
{
  std::string s = "I love programming!\n";
  DrawString1(s);
  std::cout << s;
}
void DrawString3(const std::string& text)
{
  std::cout << text;
  // we try to be evil -> FAILING compiler error:
  text = "I cause a compiler error by trying to change the original string \
               through the const reference.\n";
}
```

Practical guidelines

- Pass by value:
  - for basic type variables (int, double, bool)
- Pass by const Reference: (default for objects)
  - for objects (e.g. string, Point2f) that may NOT be modified.
- Pass by Reference:
  - for any types variables that need to be modified by the function.

Contents

- default parameters
- Function reference parameters
- return by reference
- recursive functions

Local variables of a function

- Scope:
  - Inside the block or compound statement
- Lifetime:
  - Starts at the declaration
  - The local variable is destroyed when the local block is left.
  - see stack

Return by value

- simplest and safest return type to use.
- a copy of that value is returned to the caller.
- literals, variables, or expressions (eg. x+1)

Return by reference

- Similar to pass by reference, values returned by reference must be variables (you can not return a reference to a literal or an expression).
- When a variable is returned by reference, a reference to the variable is passed back to the caller. The caller can then use this reference to continue modifying the variable, which can be useful at times. Return by reference is also fast, which can be useful when returning structs and classes.

Return by reference

- Danger: you should not return local variables to the function by reference.

```
int& doubleValue(int x)
{
  int value = x * 2;
  return value; // return a reference to value here
} // value is destroyed here
```

The program is returning a reference to a value that will be destroyed when the function returns. This would mean the caller receives a reference to garbage. Fortunately, your compiler will probably give you a warning or error if you try to do this.

Return by value or reference

When to use return by reference:

- When returning a reference argument
- When returning a large structure or class that has scope outside of the function

When to use return by value:

- When returning variables that were declared inside the function (use return by value) (local variables)
- When returning a built-in array or pointer value (use return by address)

Return by reference: example

Arguments a and b are references. Returning a reference is ok here.

```
int& GetSmallest(int& a, int& b)
{
  if(a < b) return a;
  return b;
}
```

Advantages of passing by reference:

- It allows a function to change the value of the argument, which is sometimes useful. Otherwise, const references can be used to guarantee the function won't change the argument.
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- References can be used to return multiple values from a function.
- References must be initialized, so there's no worry about null values.

Disadvantages of passing by reference:

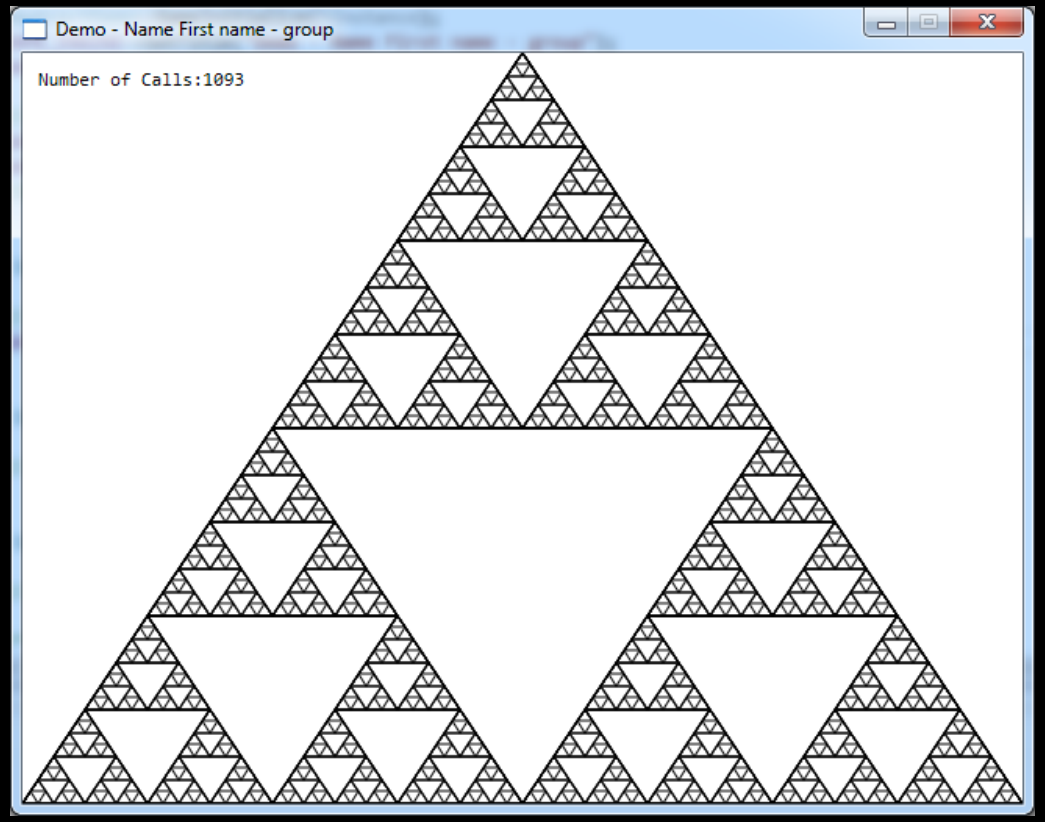
- Because a non-const reference cannot be made to an rvalue (e.g. a literal or an expression), reference arguments must be normal variables.
- It can be hard to tell whether a parameter passed by non-const reference is meant to be input, output, or both. Judicious use of const and a naming suffix for out variables can help.
- It's impossible to tell from the function call whether the argument may change. An argument passed by value and passed by reference looks the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.

Contents

- default parameters
- Function reference parameters
- return by reference
- recursive functions

Recursion

- When a function calls itself
- Demo



Recursion

- Example:

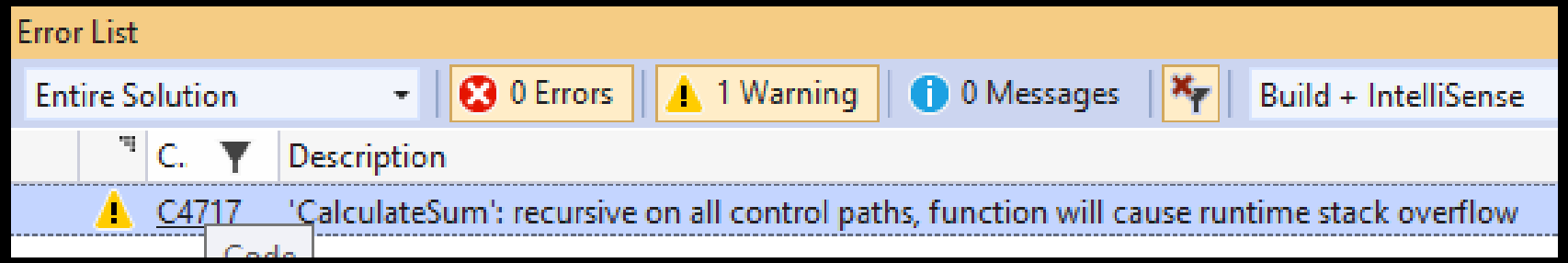
```
int main()
{
    int sum = CalculateSum(4);
}
int CalculateSum(int number)
{
    return number + CalculateSum(number-1);
}
```

Recursion

- Example:

```
int main()
{
    int sum = CalculateSum(4);
}
int CalculateSum(int number)
{
    return number + CalculateSum(number-1);
}
```

- Problem!!



Recursion

- Example:

```
int main()
{
    int sum = CalculateSum(4);
}
int CalculateSum(int number)
{
    if (number == 1) return 1;
    return number + CalculateSum(number-1);
}
```

Ok!

Recursion

```
int Sum(int number)
{
    if (number == 1) return 1;
    return number + Sum(number - 1);
}
```

Analyses:

- Sum(1) = 1
- Sum(2) = 2 + 1 = 3
- Sum(3) = 3 + Sum(2) = 3 + 2 + 1 = 6
- Sum(4) = 4 + Sum(3) = 4 + 3 + Sum(2) = 4 + 3 + 2 + 1 = 10
- Sum(5) = ...

Recursion

- When a function calls itself
- Difficult
- Hard to debug
- Danger for Infinite loop!!
- Typical use: Binary tree search, directory search
- Conclusion: use it only when there is no alternative solution to solve the problem.

References

- http://www.learncpp.com/cpp-tutorial/71-function-parameters-and-arguments/
- https://www.learncpp.com/cpp-tutorial/7-11-recursion/