EDAF30 – Programming in C++

2. Introduction. More on function calls and types.

Sven Gestegård Robertz

Computer Science, LTH

2018

Outline

Functions
Function calls

The semantics of function argument passing is the same as copy initialization: *(Same as for primitive types in Java)*

- In a function call, the *values of the arguments* are
  - type checked, and
  - with implicit type conversion (if needed)
  - copied to the function parameters
- Example: with a function `double square(double d)`

```
double s2 = square(2);        // 2 is converted to double
                              // double d = 2;

double s3 = square("three"); // error
                              // double d = "three";
```


Functions
Function overloading

- Overloading ("överlagring")

```
void print(int);
void print(double);
void print(std::string);

void user()
{
    print(42);      // calls print(int);
    print(1.23);    // calls print(double);
    print(4.5f);    // calls print(double);
    print("Hello") // calls print(std::string);
}
```

- Cannot differ only in return type
- Must not be ambiguous

- Default arguments (sometimes) similar to overloading
  - `void print(int x, int max_width=20);`
  - The rules are complex. *Only use for trivial cases*
  - Risk of ambiguity if combined with overloading

Functions
Call - ambiguity

- With overloaded functions, the compiler selects "the best" function (after implicit type conversion)
- If two alternatives are "equally good matches " it is an error

```
void print2(int, double);
void print2(double, int);

void user()
{
    print2(0, 0);    // Error! ambiguous
}
```

- and also (with `print()` from last slide)

```
long l = 17;
print(l);        // Error! print(int) or print(double)?
```


Functions
Rule of thumb

Factor your code into small functions to

- give names to activities and document their dependencies
- avoid writing specific code in the middle of other code
- facilitate testing

- A function should perform a single task
- Keep functions as short as possible
- Rule of thumb
  - Max 24 lines
  - Max 80 columns
  - Max 3 block levels
  - Max 5–10 local variables
  - Inversely proportional to complexity

Call by value and call by reference
Call by value(*värdeanrop*)

In a 'normal' function call, the values of the arguments are copied to the formal parameters (which are local variables)

Example: swap two integer values

```
void swap(int a, int b)
{
   int tmp=a;
   a = b;
   b = tmp;
}
```

...and use:

```
int x = 2;
int y = 10;

swap(x, y);

cout << x ", " << y << endl;
```

2,10 x and y are not changed

Call by value and call by reference
Call by reference(*referensanrop*)

Use *call by reference* instead of *call by value*:

Example: swap two integer values

```
void swap(int& a, int& b)
{
   int tmp=a;
   a = b;
   b = tmp;
}
```

...and use:

```
int x = 2; int y = 10;

swap(x, y);
```

Here, *references* to the arguments are used , and the values are actually swapped.

References

- A reference is *an alias* for a variable

The call `swap(x,15);` gives the error message

```
invalid initialization of non-const reference of type "int&"
from an rvalue of type 'int'
```

NB! The argument for a reference parameter must be an *lvalue*

Data types
Pointers, Arrays and References

- References
- Pointers (similar to Java references)
- Arrays ("built-in arrays"). Similar to Java arrays of primitive types

Pointers

Similar to references in Java, but

- a pointer is the *memory address of an object*
- a pointer *is an object* (a C++ reference is not)
  - can be assigned and copied
  - has an address
  - can be declared without initialization, but then it gets an *undefined value* , as do other variables
- four possible states
  1. point to an object
  2. point to the address immediately past the end of an object
  3. point to nothing: `nullptr`. Before C++11: `NULL`
  4. invalid
- can be used as an integer value
  - arithmetic, comparisons, etc.

Be very careful!

Pointers
Syntax, operatorers * and &

- In a *declaration*:
  - prefix *: "pointer to"
    - `int *p;` : p is *a pointer to an int*
    - `void swap(int*, int*);` : *function taking two pointers*
  - prefix &: "reference to"
    - `int &r;` : r is *a reference to an* **int**
- In an *expression*:
  - prefix *: dereference, "contents of"
    - `*p = 17;` *the object that p points to* is assigned 17
  - prefix &: "address of", "pointer to"

```
int x = 17;
int y = 42;

swap(&x, &y);
```

Call `swap(int*, int*)` with *pointers to* `x` *and* `y`

Pointers
Be careful with declarations

Advice: One declaration per line

```
int *a;     // pointer to int
int* b;     // pointer to int
int c;      // int

int* d, e;  // d is a pointer, e is an int
int* f, *g; // f and g are both pointers
```

Choose a style, either `int *a` or `int* b`, and be consistent.

References

References are similar to pointers, but

- A reference is *an alias to* a variable
  - cannot be changed (*reseated* to refer to another variable)
  - must be initialized
  - is not an object (has no address)
  - Dereferencing does not use the operator *
    - Using a reference *is* to use the referenced object.

Use a reference if you don't have (a good reason) to use a pointer.

- E.g., if it may have the value `nullptr` (*"no object"*)
- or if you need to change("reseat") the pointer
- More on this later.

Pointers and references
Call by pointer

In some cases, a *pointer* is used instead of a *reference* to "call by reference:

Example: swap two integers

```
void swap2(int* a, int* b)
{
  if(a != nullptr && b != nullptr) {
    int tmp=*a;
    *a = *b;
    *b = tmp;
  }
}  ...  and use:           int x, y;
                           ...
                           swap2(&x, &y);
```

NB!:

- a pointer can be `nullptr` or uninitialized
- dereferencing such a pointer gives *undefined behaviour*

Pointers and references

Pointer and reference versions of `swap`

```
// References
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
// Pointers
void swap(int* pa, int* pb)
{
  if(pa != nullptr && pb != nullptr) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
  }
}
```

```
int m=3, n=4;
swap(m,n);    Reference version is called

swap(&m,&n); Pointer version is called
```

NB! Pointers are *called by value*: *the address* is copied

Pointers and references

Pointer and reference versions of `swap`

```
// References
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
// Pointers
void swap(int* pa, int* pb)
{
  if(pa != nullptr && pb != nullptr) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
  }
}
```

```
int m=3, n=4;
swap(m,n);    Reference version is called

swap(&m,&n); Pointer version is called
```

NB! Pointers are *called by value*: *the address* is copied

Arrays ("C-arrays", "*built-in arrays*")

- A sequence of values of the same type (homogeneous sequence)
- Similar to Java for primitive types
  - but *no safety net* – difference from Java
  - an array does not know its size – the programmer's responsibility
- *Can contain elements of any type*
  - Java arrays *can only contain references* (or primitive types)
- Can be a local (or member) variable (Difference from Java)
- Is declared `T a[size];` (Difference from Java)
  - The size must be *a (compile-time) constant.* (Different from C99 which has VLAs)

Arrays
Representation in memory

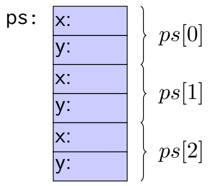
The elements of an array can be of any type

- Java: only primitive types or a reference to an object
- C++: an object or a pointer

Example: array of `Point`

```
class Point{
    int x;
    int y;
};

Point ps[3];
```



Important difference from Java: no fundamental difference between built-in and user defined types.

Data types
C strings

- C strings are `char[]` that are *null terminated*.
  Example: `char s[6] = "Hello";`

s:	'H'	'e'	'l'	'l'	'o'	'\0'

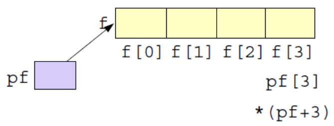
NB! A *string literal* is a C-style string (not a `std::string`)
The type of `"Hello"` is `const char[6]`.

Pointers and arrays

Arrays are accessed through pointers

```
float f[4];          // 4 floats
float* pf;           // pointer to float

pf = f;              // same as = &f[0]
float x = *(pf+3);  // Alt. x = pf[3];
x = pf[3];           // Alt. x = *(pf+3);
```



Pointers and arrays
What does array indexing really mean?

The expression `a[b]` is equivalent to `*(a + b)` (and, thus, to `b[a]`)

Definition

For a pointer, `T* p`, and an integer `i`, the expression `p + i` is defined as `p + i * sizeof(T)`

That is,

- `p+1` points to the address after the object pointed to by `p`
- `p+i` is an address i *objects after* `p`.

Example: confusing code (Don't do this)

```
int a[] {1,4,5,7,9};

cout << a[2] << " == "<< 2[a] << endl;
```

```
5 == 5
```


Pointers and arrays
Function calls

Function for zeroing an array

```
void zero(int* x, size_t n) {
   for (int* p=x; p != x+n; ++p)
      *p = 0;
}
    ...
    int a[5];

    zero(a,5);
```

- *The name of an array variable* in an expression is interpreted as *"a pointer to the first element"*: *array decay*
- `a` ⇔ `&a[0]`
- arrays cannot be copied (passed by value)

Array subscripting

```
void zero(int x[], size_t n) {
  for (size_t i=0; i != n; ++i)
    x[i] = 0;
}
```

- In function parameters `T a[]` *is equivalent to* `T* a`. (Syntactic sugar)
- `T*` is more common

Two types from the standard library
Alternatives to C-style arrays

Do not use built-in arrays unless you have (a strong reason) to.
Instead of

- `char[]` – Strings – use `std::string`
- `T[]` – Sequences – use `std::vector<T>`

More like in Java:

- more functionality – *"behaves like a built-in type"*
- safety net

Strings: `std::string`

`std::string` has operations for

- assigning
- copying
- concatenation
- comparison
- input and output (<< >>)

and

- knows its size

Similar to `java.lang.String` *but is mutable*.

Sequences: `std::vector<T>`

A `std::vector<T>` is

- an ordered collection of objects (of the same type, `T`)
- every element has an index

which, in contrast to a built-in array

- knows its size
  - `vector<T>::operator[]` does no bounds checking
  - `vector<T>::at(size_type)` throws out_of_range
- can grow (and shrink)
- can be assigned, compared, etc.

Similar to `java.util.ArrayList`

Is a *class template*

Example: `std::string`

```
#include <iostream>
#include <string>
using std::string;
using std::cout;
using std::endl;

string make_email(string fname,
                  string lname,
                  const string& domain)
{
    fname[0] = toupper(fname[0]);
    lname[0] = toupper(lname[0]);
    return fname + '.' + lname + '@' + domain;
}

void test_string()
{
    string sr = make_email("sven", "robertz", "cs.lth.se");

    cout << sr << endl;
}
```

```
Sven.Robertz@cs.lth.se
```


Example: `std::vector<int>` initialisation

```
void print_vec(const std::string& s, const std::vector<int>& v)
{
    std::cout << s << " : " ;
    for(int e : v) {
        std::cout << e << " ";
    }
    std::cout << std::endl;
}
void test_vector_init()
{
    std::vector<int> x(7);
    print_vec("x", x);

    std::vector<int> y(7,5);
    print_vec("y", y);

    std::vector<int> z{1,2,3};
    print_vec("z", z);
}
```

```
x: 0 0 0 0 0 0 0
y: 5 5 5 5 5 5 5
z: 1 2 3
```


Example: `std::vector<int>` assignment

```
void test_vector_assign()
{
    std::vector<int> x {1,2,3,4,5};
    print_vec("x", x);
    std::vector<int> y {10,20,30,40,50};
    print_vec("y", y);
    std::vector<int> z;
    print_vec("z", z);
    z = {1,2,3,4,5,6,7,8,9};
    print_vec("z", z);
    z = x;
    print_vec("z", z);
}
```

```
x : 1 2 3 4 5
y : 10 20 30 40 50
z :
z : 1 2 3 4 5 6 7 8 9
z : 1 2 3 4 5
```


Example: `std::vector<int>` insertion and comparison

```
void test_vector_eq()
{
    std::vector<int> x {1,2,3};
    std::vector<int> y;
    y.push_back(1);
    y.push_back(2);
    y.push_back(3);

    if(x == y) {
        std::cout << "equal" << std::endl;
    } else {
        std::cout << "not equal" << std::endl;
    }
}
```

```
equal
```

User defined types

- Built-in types (e.g., **char**, **int**, **double**, pointers, ...) and operations
  - Rich, but deliberately low-level
  - Directly and efficiently reflect the capabilites of conventional computer hardware
- User-defined types
  - Built using the *built-in types* and *abstraction mechanisms*
  - **struct**, **class** (cf. **class** i Java)
  - Examples from the standard library
    - `std::string` (cf. `java.lang.String`)
    - `std::vector`, `std::list` ... (cf. corresponding class in `java.util`)
  - **enum class**: enumeration (cf. **enum** in Java)
- A *concrete type* can behave "just like a built-in type".

Structures

The first step in building a new type is to organize the elements it needs into a data structure, a *struct*.
Exempel: Person

```
struct Person{
    string first_name;
    string last_name;
};
```

A variable of the type `Person` is created with

```
Person p;
```

but now *the member variables* have *default initialized values*.
NB! that sometimes means *undefined*

More on object initialization later.

Structures
Initialization

A function for initializing a `Person`:

```
void init_person(Person& p, const string& fn,  const string& ln)
  {
      p.first_name = fn;
      p.last_name = ln;
  }
```

A variable of type `Person`, can be created and initialized with

```
Person sven;
init_person(sven, "Sven", "Robertz");
```

- call by reference: the object `sven` is changed

Structures
Use

Now we can use our type `Person`:

```
#include <iostream>
Person read_person()
{
    cout << "Enter first name:" << endl;
    string fn;
    cin >> fn;

    cout << "Enter last name:" << endl;
    string ln;
    cin >> ln;

    Person p;
    init_person(p, fn, ln);
    return p;
}
```

- `>>` is *the input operator*
- the standard library `<iostream>`
- `std::cin` is *standard input*

Classes

Make a type behave like a built-in type
- Tight coupling of data and operations
- Often make the representation inaccessible to users

A class has
- data members ("attributes")
- member functions ("methods")
- members kan be
  - **public**
  - **private**
  - **protected**
  - like in Java

Classes
Example

```
class Person{
public:
    Person(string fn, string ln) :first_name{fn}, last_name{ln} {}
    string get_name();
    string get_initials();
private:
    string first_name;
    string last_name;
};
```

- *constructor*, like in Java
  - Creates an object and *initializes members*
  - the statements
    ```
    Person sven;
    init_person(sven, "Sven", "Robertz");
    ```
    become `Person sven("Sven", "Robertz");`

***class** and **struct** are (mostly) synonyms in C++.*

Classes
Example

```
Person read_person()
{
    cout << "Enter first name:" << endl;
    string fn;
    cin >> fn;
    cout << "Enter last name:" << endl;
    string ln;
    cin >> ln;
    return Person(fn, ln);
}

void test_read()
{
    Person p = read_person();
    cout << p.get_initials() << " : " << p.get_name() << endl;
}
```


Class definitions
Declarations and definitions of member functions

Member functions (⇔ methods in Java)

Definition of a class

```
class Foo {
public:
    int fun(int, int);    // Declaration of member function

    int times_two(int x) {return 2*x;} // ... incl definition
};
```

NB! Semicolon after class

Definition of member function (outside the class)

```
int Foo::fun(int x, int y) {
    // ...
}
```

No semicolon after function

File structure for classes

- The class definition is put in a header file (.h or .hpp)
- To avoid defining a class more than once, use *include guards*:

```
#ifndef FOO_H
#define FOO_H
//...
class Foo {
//...
};
#endif
```

- Member function definitions are put in a source file (.cc)

Declarations
Scope

A declaration introduces a *name* in a *scope*

Local scope: A name declared in a function is visible
- From the declaration
- To the end of the block (delimited by{ })
- Parameters to functions are local names

Class scope: A name is called a *member* if it is declared *in a class**. It is visible in the entire class.

Namespace scope: A named is called a *namespace member* if it is defined *in a namespace**. E.g, `std::cout`.

A name declared outside of the above is called a *global name* and is in *the global namespace*.

* outside a function, class or *enum class*.

Declarations
lifetimes

- The lifetime of an object is determined by its *scope*:
- An object
  - must be initialized (constructed) before it can be used
  - is destroyed *at the end of its scope*.
- a *local variable* only exists until the function returns
- *namespace objects* are destroyed when the program terminates
- an *object allocated with* **new** lives until destroyed with **delete**. (different from Java)
  - Manual memory management
  - **new** is not used as in Java
  - Avoid **new** except in special cases
  - more on this later

Stream I/O

- The C++ standard library contains facilities for
  - Structured I/O ("formatted I/O")
    - reading values of a certain type, T
    - overload `operator>>(istream&, T&)` and
    - `operator<<(ostream&, const T&)`
  - Character I/O ("raw I/O")
    - `istream& getline(istream&, string&)`
    - `istream& istream::getline(char*, streamsize)`
    - `int istream::get()`
    - `istream& istream::ignore()`
    - ...
- NB! `getline()` as free function and member of `istream`.
- Choose raw or formatted I/O based on your application

Suggested reading

References to sections in Lippman

Literals 2.1.3

Pointers and references 2.3

std::string 3.2

std::vector 3.3

Arrays and pointers 3.5

Classes 2.6, 7.1.4, 7.1.5, 13.1.3

Scope and lifetimes 2.2.4, 6.1.1

I/O 1.2, 8.1–8.2, 17.5.2

Next lecture
Classes

References to sections in Lippman

Classes 2.6, 7.1.4, 7.1.5

Constructors 7.5–7.5.4

(Aggregate classes) ("C structs" without constructors) 7.5.5

Operator overloading 14.1 – 14.3, 14.5 – 14.6

this and const p 257–258

inline 6.5.2, p 273

friend 7.2.1

static members 7.6

const, constexpr 2.4