Programmazione e Calcolo Scientifico

Matteo Cicuttin

Politecnico di Torino

March 19, 2024

Pitfalls of arithmetic operators: a clarification

In the third class I showed the following examples:

```
double x_bad = 1/2;              // x_bad is zero: integer division
double x_ok = 1./2.;             // x_ok is 0.5
```

I was told that the `1./2.` gave many headaches. I have a message for you:

C++ IS NOT MATLAB.

- `double x_ok` declares a **scalar**
- the operator `./` **does not exist** in C++ (see the slide about operators)
- the above initialization is read `1.  / 2.` which is equivalent to `1.0 / 2.0`

Structures

Sometimes it is needed to group together related data. For this we use `struct`s. For example:

```
struct meteo_data {
    std::string location_name;
    int         minute, hour, day, month, year;
    double      temperature;
    double      pressure;
};
```

Subsequently, to create an object of type `meteo_data`:

```
meteo_data md;
md.location_name = "Torino";
md.minute = 42;
md.hour = 21;
md.temperature = 16.3;
// and so on...
```

Passing parameters to functions: by value

We already discussed some examples of functions. To be useful, functions need parameters. Until now we passed parameters **by value**.

```
int f(int x) {
    x = x+1;
    return x;
}

int main(void) {
    int y = 5;
    f(y);
    std::cout << y << std::endl;
}
```

The `x` in function `f` is like a local variable

- during the execution of `f`, the variable `x` is a **copy** of `y`
- if you modify `x`, then `y` is not touched
- in general this is ok, but with large objects (e.g. `struct`s) copies are expensive (`sizeof(meteo_data)` is large)

Passing parameters to functions: by const reference

If you need to pass a large object as a parameter but you don't want to pay the price of a copy, pass by **const reference**

```
using namespace std;

void print(const meteo_data& md) {
  cout << md.location_name;
  cout << ": " << md.temperature;
  cout << "\n";
  md.temperature = 42.0; // error!
  // md is const, no writes allowed
}

int main(void) {
  meteo_data md;
  // ... code to fill meteo_data
  print(md);
}
```

The `md` in `print` is a const reference to `md` in `main`

- no copies are made, and:
- when you read `md` in `print`, you are actually reading `md` in `main`
- modifications to `md` are not allowed as the parameter is qualified `const`
- if you need to pass a struct, pass it by const reference

Passing parameters to functions: by reference

What if we remove the `const` from the parameter?

```
void read_sensors(meteo_data& md) {
  md.temperature = 21.0; // ok
  md.pressure = 1013.5; //ok
}

int main(void) {
  meteo_data md;
  read_sensors(md);
  print(md);
}
```

The `md` in `read_sensors` is a reference to `md` in `main`

- no copies are made, and:
- when you write `md` in `read_sensors`, you are actually writing `md` in `main`
- modifications to the `read_sensors` parameter `md` are allowed and happen in something outside the function

Passing parameters to functions: by const pointer

Pass by const reference can be **simulated** with pointers:

```
using namespace std;

void print(const meteo_data* md) {
  cout << (*md).location_name;
  cout << ": " << (*md).temperature;
  cout << "\n";
  (*md).temperature = 42.0; // error!
  // md is const, no writes allowed
}

int main(void) {
  meteo_data md;
  // ... code to fill meteo_data
  print(&md); // notice the &
}
```

The `md` in `print` is a const pointer to `md` in `main` in `main`

- `md` in `print` contains **the address** of `md` in `main`
- to access the data you need to dereference the pointer: we discussed this in the previous class
- modifications to the pointed data are not allowed because the parameter is qualified `const`
- for now prefer pass by const reference: when we'll need to pass a pointer I'll tell you

Passing parameters to functions: by pointer

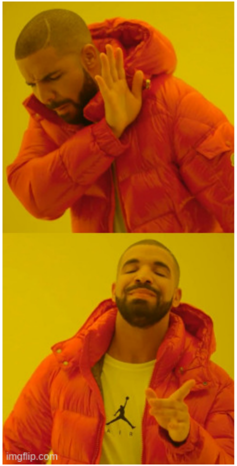
Pass by reference can be **simulated** with pointers:

```
void read_sensors(meteo_data* md) {
    (*md).temperature = 21.0; // ok
    (*md).pressure = 1013.5; //ok
}

int main(void) {
    meteo_data md;
    read_sensors(&md); // notice the &
    print(&md); // notice the &
}
```

The `md` in `read_sensors` is a pointer to `md` in `main`

- `md` in `read_sensors` contains **the address** of `md` in `main`
- to access the data you need to dereference the pointer: we discussed this in the previous class
- modifications to the pointed data are allowed
- for now prefer pass by reference: when we'll need to pass a pointer I'll tell you


imgflip.com

*

&

Exercise

Warning!

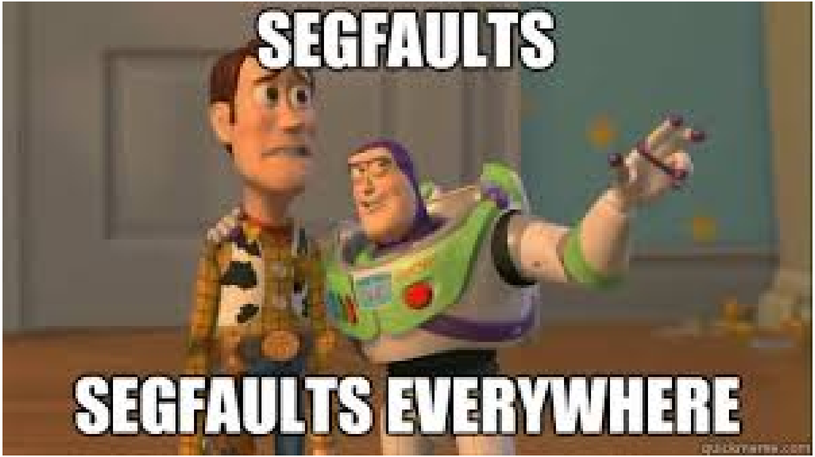
```c
void read_sensors(meteo_data* md) {
  (*md).temperature = 21.0;
  (*md).pressure = 1013.5;
}

int main(void) {
  meteo_data md;
  read_sensors(&md);
}
```

```c
void read_sensors(meteo_data* md) {
  (*md).temperature = 21.0;
  (*md).pressure = 1013.5;
}

int main(void) {
  meteo_data* md;
  read_sensors(md);
}
```

One of the programs is incorrect and if you run it, it explodes. Tell me which one and why.


SEGFAULTS
SEGFAULTS EVERYWHERE

Pointers vs. references

Pointers can point anywhere, even at invalid memory. On the other hand, references can't be invalid. If you don't initialize a reference where you declare it, the program is invalid and does not compile.

```
int main(void) {
  meteo_data md;        // OK: this is a variable
  read_sensors(md);     // allocated on the stack
}

int main(void) {
  meteo_data* md;       // uninitialized pointer: compiles
  read_sensors(md);     // but points to invalid memory
}

int main(void) {
  meteo_data& md;       // uninitialized reference: does not
  read_sensors(md);     // compile, it is an invalid program
}
```

Notation

Warning!

```c
void read_sensors(meteo_data* md) {
  (*md).temperature = 21.0;
  (*md).pressure = 1013.5;
}
```

```c
void read_sensors(meteo_data* md) {
  *md.temperature = 21.0;
  *md.pressure = 1013.5;
}
```

The program on the left is correct, the one on the right is wrong:

- `(*md).temperature` means "dereference `md` and access `temperature`"
- `*md.temperature` would be equivalent `*(md.temperature)`, but:
  - `md` is not a struct, so you can't use "`.`"
  - even if you could, `temperature` is a `double`, not a pointer, so you can't dereference it

A new operator: `->`

Instead of writing `(*md).temperature` we normally write `md->temperature`.

A last detail

Also pointers can be passed by reference and by pointer:

```cpp
void f(const double*& ptr) {
  // function body
}

void f(double*& ptr) {
  // function body
}
```

```cpp
void f(const double** ptr) {
  // function body
}

void f(double** ptr) {
  // function body
}
```

To see it better we create an **alias** of `double*` with the `using` keyword:

```cpp
using dptr_t = double*;
void f(const dptr_t& ptr) {
  // function body
}
```

```cpp
using dptr_t = double*;
void f(const dptr_t* ptr) {
  // function body
}
```

After all, pointers are variables like all the others...

Structures and Methods

An **attribute** is a "variable belonging to a struct". A **method** is a "function belonging to a struct".

```
struct mystruct {
  int attribute;
  void method(int x);
};

void mystruct::method(int x) {
  attribute = x;
}

mystruct ms;
ms.method(42);
// ms.attribute becomes 42
```

A method is executed on a specific **instance** of our structure. When we write

```
ms.method(42);
```

think it as a fancy way of writing

```
method(ms, 42);
```

where `method` is

```
void method(mystruct& ms, int x) {
  ms.attribute = x;
}
```

A concrete example: a stack

As we said, a type not only defines the range of values for an object, but also operations on it.

A **stack** is a Last In First Out data structure. Four operations:

- push: put an object on the top of the stack
- pop: take an object from the top of the stack
- empty: check if the stack is empty
- full: check if the stack is full

Let's implement it.

Structures and Methods

```
#define STACK_SIZE 8

struct stack {
  int       data[STACK_SIZE];
  int       top;

            stack();
  void      push(int value);
  int       pop();
  bool      empty();
  bool      full();
};
```

Our stack data structure can be modeled with a struct:

- STACK_SIZE is a compile time constant
- data holds the elements we put into the stack
- top tracks how many elements we have into the stack
- Methods with the same name of the struct are called **constructors** and they are called automatically to initialize the objects

Constructor

A **constructor** is a method that gets called automatically when a new stack is created.

```
// Constructor: gets called automatically
// when we instantiate a new stack

stack::stack() {
  top = 0;
}

stack s1; // We declare new stack variables.
stack s2; // The constructor is called automatically
stack s3; // and implicitly on each object
```

After the declaration of `s1`, value of `s1.top` is zero. The same for `s2` and `s3`. It means that the stack is empty.

The role of the constructor is to bring an object in its initial, valid state.

The stack operations

```
// put something on top of stack
void stack::push(int value) {
  if (top < STACK_SIZE)
    data[top++] = value;
}
```

```
// take something from top of stack
int stack::pop() {
  if (top > 0)
    return data[--top];

  return 0;
}
```

```
// check if stack is full
bool stack::full() {
  return (top == STACK_SIZE);
}
```

```
// check if stack is empty
bool stack::empty() {
  return (top == 0);
}
```

The `::` operator is called scope resolution operator. It is used to define a function outside the struct it belongs to.

$\Longrightarrow$ stack program demo

Classes

Usually you want the user to interact with the stack only via its operations. You don't want the user to mess with the internal state.

```
#define STACK_SIZE 8

class stack {
  int     data[STACK_SIZE];
  int     top;
public:
          stack();
  void    push(int value);
  int     pop();
  bool    empty();
  bool    full();
};
```

Internal state should be hidden from the user. The program

```
stack s;
s.top = 70;
```

should be invalid.

- Classes are like structs, but by default all its members and attributes are private (not accessible by the user)
- Only members and attributes in the public section are accessible by the user

Teaser: a generic stack implementation

```
#define STACK_SIZE 8

template<typename T>
class stack {
  T     data[STACK_SIZE];
  int   top;
public:
        stack();
  void  push(const T& value);
  T     pop();
  bool  empty();
  bool  full();
};
```

What if I want a stack of `double`s? Or of `meteo_data`?

- impossible with our original implementation
- in C++ is possible to parametrize the implementation on the type, this is called **generic programming**
- read `template<typename T>` as $\forall T$

```
stack<double>     stk_dbl; // stack of doubles
stack<meteo_data> stk_md; // stack of meteo_data
```