C++: variables

ens-lal

2013

Outline

- declaring variables
- stack allocation & life-time
- dynamic allocation (heap allocation)

Declaring variables

The general syntax to declare a new variable is:

```
<qualifier(s)> <modifier(s)> <type> <variable name>
```

`type` and `variable name` are mandatory
e.g:

```
int index;
unsigned int counter;
const std::string name = "Bob";
```

builtins

- `char`: integer type encoded on 1 `byte` $[-128; 127]$
- `short`: short integer encoded on 2 `bytes` $[-32768; 32767]$
- `int`: integer encoded on 4 `bytes`
- `long`: integer encoded on 8 `bytes`
- `float`: real, simple precision (4 `bytes`)
- `double`: real, double precision (8 `bytes`)
- `bool`: boolean, only 2 possible values (`true`|`false`)

modifiers

- modifiers modify the type they are applied to.
- `signed`, `unsigned`, `long`, and `short` can be applied to integer base types. In addition, `signed` and `unsigned` can be applied to `char`, and `long` can be applied to `double`.

modifiers

- modifiers modify the type they are applied to.
- `signed`, `unsigned`, `long`, and `short` can be applied to integer base types. In addition, `signed` and `unsigned` can be applied to `char`, and `long` can be applied to `double`.

Example:

```
unsigned short us; // [     0; 65535]
         short  s; // [-32768; 32767]
  signed short ss; // [-32768; 32767]
```

qualifiers

Qualifiers provide additional information about the variables they precede.
Example:

```
const int a = 4; // 'a' can NOT be modified anymore
```

enum type

```
enum direction { north, east, south, west }; // type decl.
     direction wind = south;                 // var. decl.
```

The compiler associates to each `direction` an integer value, starting at 0 (by default.)

void type

`void` is a pseudo-type meaning "empty". Used as a pointer type (to point at anything) or as a function return value, for functions returning nothing.

arrays

```
char   c[10];      // array of 10 characters
double d[10][20]; // a 2d-array of doubles
```

Warning: indices start at 0.
Warning: C/C++ arrays are "row-major". FORTRAN ones were "column-major".

pointer type

- C and C++ provide a means to access memory addresses under which variables are stored.
- A pointer is a variable allowing to store and manipulate these addresses.
- the address of a variable a can be obtained by &a.
- access to the data stored at the address pa can be obtained by dereferencing pa: *pa.

```
int *pa;            // 'pa' is a pointer to a var. of type int
int a = 4;          // 'a' is an integer initialized w/ val. 4

pa = &a;            // initialization of 'pa' w/ addr. of 'a'
std::cout << *pa << std::endl; // => displays '4' on screen.
```

- a pointer can (and always should) be initialized to a null value:

```
int *pa = NULL;
```

Lifetime of a variable

- Reminder: in C and C++, each and every variable must be declared.
- the portion of code where a variable is "known", is called the scope of that variable. The scope starts at the declaration line and ends with the block in which the variable was defined (marked by a }}

```
void fct() {
  int i = 42;        // 'i' is known 'til the end of the fct
  for (i=0; i<10; ++i) {
     int j = 2*i;  // 'j' is known 'til the end of the for-loop
     std::cout << j << std::endl;
  }  // <-- 'j' is "destroyed" here
} // <----- 'i' is "destroyed" here
```

Lifetime of a variable (cont'd)

- the declaration of a local variable `n` may hide declarations of other variables called `n`:
  - in enclosing scopes
  - at global scope
  - data members of the class if the function is a function member
    - ⋆ whence the usefulness of following a naming convention for data members.
- special case: a static variable is known/alive during the execution of the whole program.

Dynamic memory management: `new` and `delete`

- dynamic memory allocation allows to free the programmer from the rules of the allocation on the stack.
- but then, the programmer must manage herself the memory via:
  - pointers
  - new and delete operators to (resp.) allocate and de-allocate memory resources
- the new operator instantiates an object: it reserves enough memory to store that new object, calls the constructor to initialize that memory region and then returns the memory address of this region.
- the delete operator calls the destructor on that memory region and releases the memory back to the operating system.
- the delete and new operators can be applied on the builtin types as well as on the complex types (`classes` and `structs`)

Summary

builtins

```
// heap allocation
int *p = new int;
*p = 421;          // 'p' points to an int. init'd to 421
delete p;

// stack allocation
int i = 421;
```

Summary - II

array

```
// heap allocation
float *arr = new float[5]; // array of 5 floats
for (int i=0; i<5; ++i) { arr[i] = 1.4 * i; }
delete[] arr;

// stack allocation
float arr[5];
for (int i=0; i<5; ++i) { arr[i] = 1.4 * i; }
```

classes - heap allocation

```
Circle *c0 = new Circle;  // default c-tor
delete c0;
Circle *c1 = new Circle(x, y); // c-tor w/ params
c1->move(10, 20);
delete c1;

// array of circles
Circle *arr = new Circle[10]; arr[2].move(10,20);
delete[] arr;
```

classes - stack allocation

```
Circle c0; // default c-tor
Circle c1(x, y); // c-tor w/ params
c1.move(10, 20);
// array of circles
Circle arr[10]; arr[2].move(10, 20);
```

Questions ?

Questions ?