EDAF50 – C++ Programming

2. Pointers. Arrays. User defined types.

Sven Gestegård Robertz
Computer Science, LTH

2019

Outline

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" (*pointer → object*)
    `*p = 17;` *the object that p points to* is assigned 17
  - prefix &: "address of", "pointer to" (*object → pointer*)

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

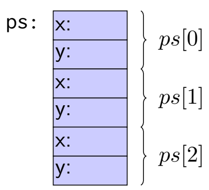
The elements of an array can be of any type

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

Example: array of `Point`

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

Point ps[3];
```



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

Multidimensional arrays

multi-dimensional arrays

- Does not (really) exist in C++
  - are arrays of arrays
  - Look like in Java
- Java: array of *references to arrays*
- C++: two alternatives
  - Array of arrays
  - Array of *pointers* (*to the first element of an array*)

Multi-dimensional arrays

Initializing a matrix with an initializer list:

3 rows, 4 columns

```
int a[3][4] = {
  {0, 1, 2, 3} ,   /*  initializer list for row 0 */
  {4, 5, 6, 7} ,   /*  initializer list for row 1 */
  {8, 9, 10, 11}   /*  initializer list for row 2 */
};
```

Instead of nested lists one can write the initialization as a single list:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

- Multi-dimensional arrays are stored like an array in memory.
- The dimension *closest to the name* is the size of the array
- The remaining dimensions belong to the element type

Multi-dimensional arrays
Representation of arrays in memory

An array `T array[4]` is represented memory by a sequence of four elements of type `T`: | T | T | T | T |

An `int[4]` is represented as

| int | int | int | int |

Thus, `int[3][4]` is represented as

| int | int | int | int | int | int | int | int | int | int | int | int |

Data types
C strings

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

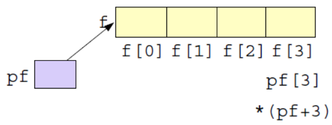
s:	'H'	'e'	'l'	'l'	'o'	'\0'

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]

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

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

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* is destroyed when the function returns
- a *member variable* is destroyed when the object is destroyed
- *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

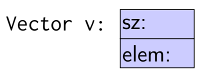
User defined types

- 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

Example: a vector of doubles

```
struct Vector {
  int sz;
  double* elem;
};
```



A variable of the type `Vector` can be created with

```
Vector v;
```

but now `v.sz` and the pointer `v.elem` are uninitialized.

To be useful, we must give `elem` som elements to point to.

Structures
Initialization

A function for initializing a `Vector`:

```
void vector_init(Vector& v, int s)
{
  v.elem = new double[s];
  v.sz = s;
}
```

A variable of type `Vector`, with size 10, can be created with

```
Vector vec;
vector_init(vec, 10); //call-by-reference: vec is changed
```

- the operator **new** allocates an object on *the heap* ("the free store")
- objects on the heap live until removed using **delete**
- more on (better alternatives to) this later

Structures
Representation

```
struct Vector {
  int sz;
  double* elem;
};
void vector_init(Vector& v, int s)
{
  v.elem = new double[s];
  v.sz = s;
}

void test()
{
   Vector vec;
   vector_init(vec, 5);
   vec.elem[2] = 7;
}
```

Structures
Use

Now we can use our `Vector`:

```
#include <iostream>
double read_and_sum(int s)
{
    Vector v;              // create Vector object
    vector_init(v,s);      // initialize v with size s
    for(int i=0; i!=s; ++i) {
        std::cin >> v.elem[i];
    }

    double sum{0};
    for(int i=0; i!=s; ++i) {
        sum += v.elem[i];
    }

    return sum;
}
```

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

Structures
Access of `struct` members

```
Vector v(10);

Vector& rv = v;

Vector* pv = &v;

...

int i = v.sz;      // access via name (of variable)

int j = rv.sz;     // access via reference (alias for name)

int k = pv->sz;    // access via pointer
```


Access of members through pointers
The operator ->

For a pointer `p`, we can express
"The member `x` in the object `p` points to" in two ways:

- `(*p).x`
- `p->x`

Classes

- Make a user-defined type behave like "a real type"
- Tight coupling between operations and the data representation
- Often: make the representation inaccessible to users

A class can have

- data members ("attributes")
- member functions ("methods")
- type members
- members can be
  - **public**
  - **private**
  - **protected**
  - like in Java

Classes
Example

```
class Vector{
public:
    Vector(int s) :elem{new double[s]}, sz{s} {}   // constructor
    double& operator[](int i) {return elem[i];}    // subscripting
    int size() {return sz;}
private:
    double* elem;
    int sz;
};
```

- *constructor*, like in Java
  - Creates an object and *initializes members*
  - the statements `Vector vec; vector_init(vec, 5);` become `Vector vec(5);`
  - *constructor body* often empty
- *operators* can be overloaded, e.g. `operator[](int)`
  - `vec.elem[2]` becomes `vec[2]`
  - The representation is not accessible (`elem` is **private**)

Classes
Example

```
double read_and_sum(int s)
{
    Vector v(s);  // Create and initialize a Vector of size s
    for(int i=0; i!=v.size(); ++i) {
        std::cin >> v[i];
    }

    double sum{0};
    for(int i=0; i!=v.size(); ++i) {
        sum += v[i];
    }

    return sum;
}
```


Class definitions
Member functions: declarations and definitions

Member functions (⇔ “methods” in Java)

Definition of class

```
class Foo {
public:
    int fun(int, int);       // Declaration of member function
    int get_x() {return x;} // ... incl definition (inline)
    ...
private:
    int x;
};
```

NB! Semicolon after class definition

Definition of member function (outside the class)

```
int Foo::fun(int x, int y) {
    return 3*x + 4*y;
}
```

No semicolon after function definition

Classes
Resource management

- *RAII Resource Acquisition Is Initialization*
- An object is initialized by a *constructor*
  - Allocates the needed resources
- When an object is destroyed, its *destructor* is executed
  - Free resources owned by the object
  - In the Vector example: the array pointed to by elem

```
class Vector{
  public:
  Vector(int s) :elem{new double[s]}, sz{s} {}  // constructor
  ~Vector() {delete[] elem;}   // destructor, delete the array
  ...
};
```

Manual memory management

- Objects allocated with **new** must be freed with **delete**
- Objects allocated with **new**[] must be freed with **delete**[]
- otherwise, the program has a *memory leak*
- (much) more on this later

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
```


Data types
Two kinds of constants

- A variable declared **`const`** must not be changed(`final` in Java)
  - Roughly:"I promise not to change this variable."
  - Is checked by the compiler
  - Use when specifying function interfaces
    - A function that does not change its (reference) argument
    - A member function ("method") that does not change the state of the object.
  - Important for function overloading
    - T and **`const`** T are different types
    - One can overload **`int`** `f(T&)` and **`int`** `f(`**`const`** `T&)` (for some type T)
- A variable declared **`constexpr`** must have a value that can be computed at compile time.
  - Use to specify constants
  - Introduced in C++-11

Functions can be `constexpr`

- Means that they can be computed at compile time if the arguments are **`constexpr`**

example:

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

void test_constexpr_fn()
{
    char matrix[square(4)];

    cout << "sizeof(matrix) = " << sizeof(matrix) << endl;
}
```

Without **`constexpr`** the compiler gives the error

```
error: variable length arrays are a C99 feature
```

const and pointers

const modifies everything to the left (exception: if **const** is first, it modifies what is directly after)

Example

```
    int* ptr;
const int* ptrToConst;  //NB! (const int) *
int const* ptrToConst,  // equivalent, clearer?

int* const  constPtr;   // the pointer is constant

const int* const constPtrToConst; // Both pointer and object
int const* const constPtrToConst; // equivalent, clearer?
```

Be careful when reading:

```
char *strcpy(char *dest, const char *src);
```

(const char)*, not const (char*)

const and pointers
Example:

```
void Exempel( int* ptr,
              int const * ptrToConst,
              int* const constPtr,
              int const * const constPtrToConst )
{
    *ptr = 0;                    // OK: changes the value of the object
    ptr  = nullptr;              // OK: changes the pointer

    *ptrToConst = 0;             // Error! cannot change the value
    ptrToConst  = nullptr;       // OK: changes the pointer

    *constPtr = 0;               // OK: changes the value
    constPtr  = nullptr;         // Error! cannot change the pointer

    *constPtrToConst = 0;        // Error! cannot change the value
    constPtrToConst  = nullptr;  // Error! cannot change the pointer
}
```


Pointers

Pointers to constant and constant pointer

```
int k;             // int that can be modified
int const c = 100;// constant int
int const * pc;    // pointer to constant int
int *pi;           // pointer to modifiable int

pc = &c;   // OK
pc = &k;   // OK, but k cannot be changed through *pc
pi = &c;   // Error! pi may not point to a constant
*pc = 0;   // Error! pc is a pointer to const int

int* const cp = &k; // Constant pointer
cp = nullptr;       // Error! The pointer cannot be reseated
*cp = 123;          // OK! Changes k to 123
```


char[], char* och const char*
const is important for C-strings

A *string literal* (e.g., `"I am a string literal"`) is **const**.

- Can be stored in read-only memory
- `char* str1 = "Hello";` — *deprecated* in C++ – gives a warning
- `const char* str2 = "Hello";` — OK, the string is **const**
- `char str3[] = "Hello";` — str3 can be modified

Suggested reading

References to sections in Lippman

Pointers and references 2.3
Arrays and pointers 3.5
Classes 2.6, 7.1.4, 7.1.5, 13.1.3
std::string 3.2
std::vector 3.3
Scope and lifetimes 2.2.4, 6.1.1
const, constexpr 2.4
I/O 1.2, 8.1–8.2, 17.5.2
Operator overloading 14.1 – 14.3
enumeration types 19.3

Next lecture
Modularity

References to sections in Lippman

Exceptions 5.6, 18.1.1
Namespaces 18.2
I/O 1.2, 8.1–8.2, 17.5.2