

Practice Sessions

Astrophysical Simulations

Part 7: C++ sequences and pointers



Master of Science in Physics and Astronomy

2018-2019

Peter Camps

peter.camps@ugent.be

S9, 1st floor, office 110.014

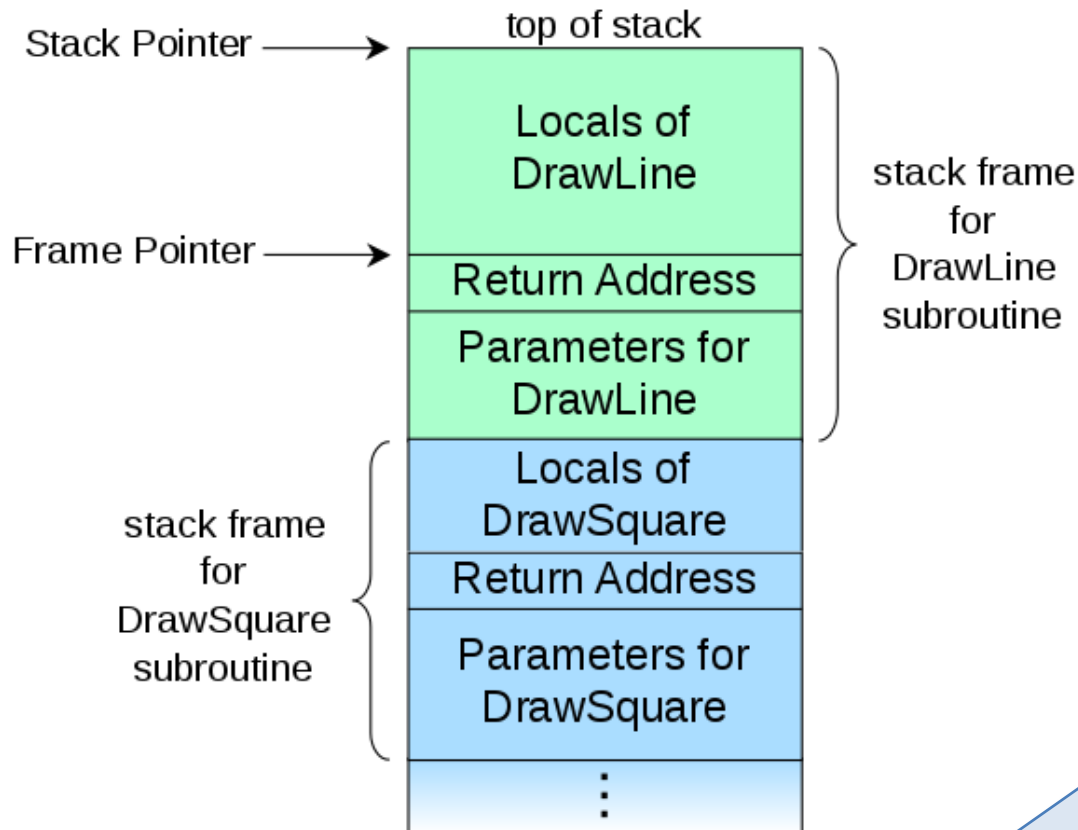
Phases of a C++ program

	Compile time	Run time
What can go wrong?	Syntax errors Type-checking errors Libraries not found	Division by zero Referencing a null pointer Open a non-existing file
If successful, what do we know?	Program was well formed It is possible to start the executable program	Executable program completes without crashing
What are the input and outputs?	In: source code, system headers and libraries Out: executable program or list of error messages	In: command line, console input, parameter files Out: console output, result files on disk

Compile-time optimization
of executable code (-O3):

- Evaluate constant expressions
- Inline eligible functions
- Remove redundant operations
- Allocate variables to registers
- Optimize loops

Run-time memory stack



The call stack

- Grows and shrinks during program execution
- Is managed via special **registers** and function call instructions
- Supports **nested** function calls and **recursion**
- Holds local variables of **fixed size** (compiler determines size of stack frame for each function)

Variable allocation on the stack incurs very little overhead

Run-time memory heap

The heap

- Is the portion of memory where **dynamically allocated** memory resides
- Can deal with block sizes that are determined at run-time

The program must

- Explicitly **allocate and free** heap memory (using *new* and *delete* in C++)
- Maintain a pointer to each allocated memory block
- Perfectly **balance** each *new* with a corresponding *delete*
 - » Neglecting to delete a memory block causes a memory leak
 - » Deleting a memory block twice causes undefined behavior (crash)

Usage considerations

- Managing memory on the heap incurs **significant overhead**
- Use when **data size** is **unknown** at compile time or is very **large** (the stack has a fairly limited size on most systems)

C++ sequence containers

Not needed:
- other STL containers
- templates, except for
using array and vector

Standard template library (STL)

- Part of the standard C++ library
- Offers a large variety of containers (sequences, hash tables, ...)
- Provided as generic template classes, where the contained data type is a parameter of the template

Sequence containers discussed here

- **std::array** – holds a **fixed** number of elements in a linear sequence
 - » Number of elements is determined at **compile time**
 - » Extremely efficient implementation for short, fixed-size sequences
- **std::vector** – holds a **variable** number of elements in a linear sequence
 - » Highly optimized implementation for variable-sized sequences
 - » Recommended choice in virtually all cases

std::array – fixed-size container

```
#include <array>
```

Include the std::array library header

```
• • •
```

Template arguments specify the element data type and array size

```
int main()
```

```
{
```

```
    array<double,4> a = { };
```

Empty initializer causes elements to be default-initialized (zero for numerics)

```
    for (double elem : a) cout << elem << ' ';
```

```
    cout << endl;
```

Initializer list should have the same length as the array size

```
    array<double,4> b = {{ 4,3,2,1 }};
```

```
    a = b;
```

Assignment copies the complete array

```
    for (double elem : a) cout << elem << ' ';
```

```
    cout << endl;
```

This form of the *for* loop iterates over all elements in a container

```
    a[0] = 5; a[3] = 6;
```

```
    for (double elem : a) cout << elem << ' ';
```

```
    cout << endl;
```

Indexing provides random access to array elements (both read and write)

```
}
```

```
0 0 0 0
4 3 2 1
5 3 2 6
```

Output of the program

std::vector – variable-size container

```
#include <vector>
```

Include the std::vector library header

```
• • •
```

```
void print(const vector<double>& v) {  
    cout << v.size() << " : ";  
    for (double elem : v) cout << elem << ' ';  
    cout << endl;  
}
```

Pass vector as constant
reference to avoid full copy

Returns the current size

Template argument specifies the
element data type

```
int main() {  
    vector<double> a;  
    print(a);
```

Default-constructed vector is empty

```
    vector<double> b = { 4, 3, 2, 1 };  
    print(b);
```

Vector is initialized with the elements
specified in the initializer list

```
    vector<double> c(5);  
    print(c);
```

Constructor argument specifies initial
number of elements, always default-
initialized (zero for numerics)

Output

```
0 :  
4 : 4 3 2 1  
5 : 0 0 0 0 0
```

std::vector – example with Vec elements

```
int main()
{
```

```
    vector<Vec> a;
    a.push_back(Vec(0,1));
    a.push_back(Vec(4,7));
    a.push_back(Vec(3,1));
```

Default-constructed vector is empty

The *push_back* function adds an element at the end of the vector

```
    for (size_t i=0; i!=a.size(); ++i) {
        a[i] /= 2;
    }
```

The size of and index into a vector is of type *size_t* (unsigned integer)

```
    cout << a.size() << endl;
    for (Vec v : a) {
        cout << v.x() << ' ' << v.y() << endl;
    }
```


This for loop iterates over all elements

Output

```
3
0 0.5
2 3.5
1.5 0.5
```


std::vector – important operations

Function	Description
Constructing	See examples on previous slides
Assignment	Fully copies the vector including size and all elements
Indexing with [index]	Accesses the element at position <i>index</i> (zero-based) without bounds checking
at(index)	Accesses the element at position <i>index</i> (zero-based) with bounds checking
size()	Returns the current number of elements in the vector
resize(n)	Resizes the container so that it contains <i>n</i> elements; extra elements are default-constructed
clear()	Removes all elements from the vector, leaving it with a size of zero
push_back(element)	Adds a new element after the current last element



at() is a bit slower than [] but is highly recommended during debugging

Passing function arguments by reference

```
void function1(vector<double> v)
{
    • • •
}
```

Pass by value

A full copy of the vector is made for every function call, adding substantial performance overhead. Any changes to the vector inside the function **do not affect** the caller's copy.

```
void function2(const vector<double>& v)
{
    • • •
}
```

Pass by constant reference

Only a reference to the vector is passed to the function (no copying overhead). The compiler will **prohibit** any changes to the vector inside the function because of the *const* qualifier.

Recommended

```
void function3(vector<double>& v)
{
    • • •
}
```

Pass by (non-constant) reference

Only a reference to the vector is passed to the function (no copying overhead). Changes to the vector inside the function **directly affect** the caller's version (since it is the same copy).

C/C++ native arrays and pointers to arrays

```
double a[4];  
double* b = new double[n];
```

Do **NOT** use
native C/C++ arrays
or pointers to arrays

- Native C/C++ arrays confusingly “decay” into a pointer to the first array element depending on the context
 - » e.g. returning a native array actually returns just a pointer to the first array element, not the contents of the array
- In low-level code such as the standard library (or in legacy code), native arrays and pointers are used to manage sequences of values
- These constructs should not be used in a higher-level program; the standard library offers far better alternatives

Benefits of standard library sequences

STL containers `std::array` and `std::vector`

- In almost all cases the STL containers are **just as efficient** as (or better than) hand-crafted alternatives using native arrays
- **Memory** is automatically **released** when an object goes out of scope
- **Assignment** (including argument passing to and returning from a function) has the expected **semantics** (the full object is copied)
- The object knows its size and can do **bounds checking** on indexed access
- There is a special form of *for* loop to iterate over elements

Pointers to objects

Pointers to objects (i.e. instances of a user-defined C++ class)

- Many object-oriented software design patterns involve the need for polymorphism and/or for interconnections between objects
- In this case, objects are constructed on the heap using the *new* operator and released back to the system using the *delete* operator
- Programmers employ specific patterns to avoid memory leaks
 - » E.g. *new* in constructor, *delete* in destructor

Pointers to objects and the corresponding software design patterns are **key components of larger software designs**

Pointers to objects

Managing memory allocation

- Always balance a *new* operation with exactly one *delete* operation
 - » A missing *delete* leaks memory, wasting an important resource (C++ has no garbage collection!)
 - » Duplicate *deletes* cause undefined (usually very nasty) behavior
- The destructor of the class where the *new* operation occurs is almost always the appropriate place for the *delete* operation
- Avoid *new* / *delete* by allocating objects on the stack where possible

Smart pointers

- C++11/14 offers smart pointers (*std::unique_ptr<>*, *std::shared_ptr<>*) that automatically delete an object when the last pointer referencing it goes out of scope



**You are welcome to use
(smart) pointers, but beware
of the intricacies involved**

Questions?