Programming in C++ - Primer

Lesson 3 - Pointers

Jakub 'Eremiell' Marek
<marekj14@fel.cvut.cz>

Silicon Hill C++ Academy

2013/11/04

Welcome!

Revisiting basics

Conditions

Revisiting basics

Conditions

- if

Revisiting basics

Conditions

- if
- switch/case

Revisiting basics

Conditions

- if
- switch/case
- ternary

Revisiting basics

Conditions

- if
- switch/case
- ternary

Loops

Revisiting basics

Conditions

- if
- switch/case
- ternary

Loops

- while

Revisiting basics

Conditions

- if
- switch/case
- ternary

Loops

- while
- do/while

Revisiting basics

Conditions

- if
- switch/case
- ternary

Loops

- while
- do/while
- for

Diving deeper

Diving deeper

Headers

Diving deeper

Headers

- system/library <>

Diving deeper

Headers

- system/library <>
- local/project ""

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Preprocessor directives

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Preprocessor directives

- include

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Preprocessor directives

- include
- define/undef

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Preprocessor directives

- include
- define/undef
- if/ifdef/ifndef/elif/else/endif

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Preprocessor directives

- include
- define/undef
- if/ifdef/ifndef/elif/else/endif

I/O Streams

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Preprocessor directives

- include
- define/undef
- if/ifdef/ifndef/elif/else/endif

I/O Streams

- formatted file input/output

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Preprocessor directives

- include
- define/undef
- if/ifdef/ifndef/elif/else/endif

I/O Streams

- formatted file input/output
- standard in: cin

Diving deeper

Headers

- system/library <>
- local/project ""
- shield

Preprocessor directives

- include
- define/undef
- if/ifdef/ifndef/elif/else/endif

I/O Streams

- formatted file input/output
- standard in: cin
- standard out: cout, cerr (error)

Diving deeper

Diving deeper

Arrays

Diving deeper

Arrays

- type name[size]

Diving deeper

Arrays

- type name[size]
- fixed size

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Overloading

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Overloading

- same name, different function

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Overloading

- same name, different function
- must have different input types

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Overloading

- same name, different function
- must have different input types
- doesn't care about names

Passing by value/reference

Passing by value/reference

Value

Passing by value/reference

Value

- we get a copy

Passing by value/reference

Value

- we get a copy
- original remains unchanged

Passing by value/reference

Value

- we get a copy
- original remains unchanged
- we use standard variables

Passing by value/reference

Value

- we get a copy
- original remains unchanged
- we use standard variables

```
int count(int);
int count(int a) {...};
int a;
count(a);
count(10);
```

Passing by value/reference

Value

- we get a copy
- original remains unchanged
- we use standard variables

Reference

```
int count(int);
int count(int a) {...};
int a;
count(a);
count(10);
```

Passing by value/reference

Value

- we get a copy
- original remains unchanged
- we use standard variables

Reference

- we get the original

```
int count(int);
int count(int a) {...};
int a;
count(a);
count(10);
```

Passing by value/reference

Value

- we get a copy
- original remains unchanged
- we use standard variables

Reference

- we get the original
- we can change it

```
int count(int);
int count(int a) {...};
int a;
count(a);
count(10);
```

Passing by value/reference

Value

- we get a copy
- original remains unchanged
- we use standard variables

Reference

- we get the original
- we can change it
- we use pointers

```
int count(int);
int count(int a) {...};
int a;
count(a);
count(10);
```

Passing by value/reference

Value

- we get a copy
- original remains unchanged
- we use standard variables

```
int count(int);
int count(int a) {...};
int a;
count(a);
count(10);
```

Reference

- we get the original
- we can change it
- we use pointers

```
int count(int*);
int count(int *a) {...};
int *a, b;
count(a);
count(&b);
```

Pointers

Pointers

address of memory, where value is saved

Pointers

address of memory, where value is saved
is a variable and takes space in memory

Pointers

address of memory, where value is saved
is a variable and takes space in memory
is a whole number with constant size of 8/16/32/64b

Pointers

address of memory, where value is saved
is a variable and takes space in memory
is a whole number with constant size of 8/16/32/64b
each architecture get some maximal memory

Pointers

address of memory, where value is saved
is a variable and takes space in memory
is a whole number with constant size of 8/16/32/64b
each architecture get some maximal memory

2^8	2^16	2^32	2^64
256B	64kB	4GB	16EB=16384PB=16777216GB

Reference & Dereference Operators

Reference & Dereference Operators

* used as data type means pointer (address)
* used as operator means dereference (data at address)

Reference & Dereference Operators

* used as data type means pointer (address)
* used as operator means dereference (data at address)
& used as data type means reference (magical address)
& used as operator means reference (address of data)

Reference & Dereference Operators

* used as data type means pointer (address)
* used as operator means dereference (data at address)
& used as data type means reference (magical address)
& used as operator means reference (address of data)

```
#include <iostream>

int main() {
    int a = 10;
    int *b = &a;
    std::cout << a << " " << *b << std::endl;
    a = 5;
    std::cout << a << " " << *b << std::endl;
    *b = 7;
    std::cout << a << " " << *b << std::endl;
    return 0;
}
```

Reference

Reference

is a special type of pointer

Reference

is a special type of pointer
is only in C++ (pointers are in C as well)

Reference

is a special type of pointer
is only in C++ (pointers are in C as well)
has to be defined at declaration

Reference

is a special type of pointer
is only in C++ (pointers are in C as well)
has to be defined at declaration
cannot be changed (the address, not the data)

Reference

is a special type of pointer
is only in C++ (pointers are in C as well)
has to be defined at declaration
cannot be changed (the address, not the data)
points to memory, you give it

Reference

is a special type of pointer
is only in C++ (pointers are in C as well)
has to be defined at declaration
cannot be changed (the address, not the data)
points to memory, you give it
behaves like standard variable

Reference

is a special type of pointer
is only in C++ (pointers are in C as well)
has to be defined at declaration
cannot be changed (the address, not the data)
points to memory, you give it
behaves like standard variable

```
#include <iostream>

int main() {
    int a = 10;
    int &b = &a;
    std::cout << a << " " << b << std::endl;
    a = 5;
    std::cout << a << " " << b << std::endl;
    b = 7;
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

Return by Parameter

Return by Parameter

function can only have one return value

Return by Parameter

function can only have one return value
sometimes we need more

Return by Parameter

function can only have one return value
sometimes we need more
make the variable outside of function

Return by Parameter

function can only have one return value
sometimes we need more
make the variable outside of function
send it in by reference

Return by Parameter

function can only have one return value
sometimes we need more
make the variable outside of function
send it in by reference

```
#include <iostream>

void count(int *a, int &b) {
    *a = *a - b;
    b = *a + 2 * b;
    return;
}

int main() {
    int a = 5, b = 3;
    std::cout << a << " " << b << std::endl;
    count(&a, b);
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

Questions?

Arrays Revisited

Arrays Revisited

arrays are pointers too!

Arrays Revisited

arrays are pointers too!
a[] is the same as *a

Arrays Revisited

arrays are pointers too!
a[] is the same as *a
it just allocates more memory for data

Arrays Revisited

arrays are pointers too!
a[] is the same as *a
it just allocates more memory for data
but you can call it by dereference

Arrays Revisited

arrays are pointers too!
a[] is the same as *a
it just allocates more memory for data
but you can call it by dereference
they don't know it's own size!

Arrays Revisited

arrays are pointers too!
a[] is the same as *a
it just allocates more memory for data
but you can call it by dereference
they don't know it's own size!
can be allocated by list

Arrays Revisited

arrays are pointers too!
a[] is the same as *a
it just allocates more memory for data
but you can call it by dereference
they don't know it's own size!
can be allocated by list

```
#include <iostream>

int main() {
    size = 5;
    int a[size];
    for (int i = 0; i < size; i++) {
        *(a + i) = i;
        std::cout << *(a + i) << std::endl;
    }
    return 0;
}
```

Multidimensional Arrays

Multidimensional Arrays

are pointers on pointers

Multidimensional Arrays

are pointers on pointers
a[][] is the same as **a

Multidimensional Arrays

are pointers on pointers
a[][] is the same as **a
you always have to name all sizes but the first

Multidimensional Arrays

are pointers on pointers
a[][] is the same as **a
you always have to name all sizes but the first
as long as you use array notation

Multidimensional Arrays


```
#include <iostream>

void printout(int array[][5], int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < 5; j++) {
            array[i][j] = i * size + j;
            std::cout << array[i][j] << std::endl;
        }
    }
}

int main() {
    size = 3;
    int a[size][5];
    printout(a, size);
    return 0;
}
```

Multidimensional Arrays


```
#include <iostream>

void printout(int **array, int sizeX, int sizeY) {
    for (int i = 0; i < sizeX; i++) {
        for (int i = j; j < sizeY; j++) {
            *(*array + i * sizeX + j) = i * sizeX + j;
            std::cout << *(*(array + i) + j) << std::endl;
        }
    }
}

int main() {
    sizeX = 3;
    sizeY = 5;
    int a[sizeX][sizeY];
    printout(a, sizeX, sizeY);
    return 0;
}
```

Multidimensional Arrays


```
#include <iostream>

void printout(int **array, int sizeX, int sizeY) {
    for (int i = 0; i < sizeX * sizeY; i++) {
        *(*array + i) = i;
        std::cout << *(*array + i) << std::endl;
    }
}

int main() {
    sizeX = 3;
    sizeY = 5;
    int a[sizeX][sizeY];
    printout(a, sizeX, sizeY);
    return 0;
}
```

Questions?

Segmentation Fault

Segmentation Fault

you'll get it (sooner or later)

Segmentation Fault

you'll get it (sooner or later)
you touched memory, you haven't allocated

Segmentation Fault

you'll get it (sooner or later)
you touched memory, you haven't allocated
bad pointer

Segmentation Fault

you'll get it (sooner or later)
you touched memory, you haven't allocated
bad pointer
already deleted pointer (NULL them!)

Segmentation Fault

you'll get it (sooner or later)
you touched memory, you haven't allocated
bad pointer
already deleted pointer (NULL them!)
bad array index (it doesn't know it's size!)

Program Arguments

Program Arguments

Just as as function can get arguments, your main can get arguments from OS

Program Arguments

Just as as function can get arguments, your main can get arguments from OS

./<yourprogram> <argument>...

Program Arguments

Just as as function can get arguments, your main can get arguments from OS

```
./<yourprogram> <argument>...
```

```
int main();
int main(int, char*[]);
```

Program Arguments

Just as as function can get arguments, your main can get arguments from OS

```
./<yourprogram> <argument>...
```

```
int main();
int main(int, char*[]);
```

commonly

```
int main(int argc, char *args[]) {...}
```

or

```
int main(int argc, char *argv[]) {...}
```

Program Arguments

Program Arguments


```
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    int i = strtol(argv[1], NULL, 10);
    int j = strtol(argv[2], NULL, 10);
    std::cout << i + j << std::endl;
    return 0;
}
```

Questions?

Memory Allocation

Memory Allocation

memory can be allocated statically (on stack) or dynamically (on heap)

Memory Allocation

memory can be allocated statically (on stack) or dynamically (on heap)
you already know static allocation

Memory Allocation

memory can be allocated statically (on stack) or dynamically (on heap)
you already know static allocation

```
int i = 5;
double d = 6.2;
int a[5] = {1, 2, 3, 4, 5};
```

Memory Allocation

memory can be allocated statically (on stack) or dynamically (on heap)
you already know static allocation

```
int i = 5;
double d = 6.2;
int a[5] = {1, 2, 3, 4, 5};
```

has it's limits

Memory Allocation

memory can be allocated statically (on stack) or dynamically (on heap)
you already know static allocation

```
int i = 5;
double d = 6.2;
int a[5] = {1, 2, 3, 4, 5};
```

has it's limits
you have to know the size on compilation time

Memory Allocation

memory can be allocated statically (on stack) or dynamically (on heap)
you already know static allocation

```
int i = 5;
double d = 6.2;
int a[5] = {1, 2, 3, 4, 5};
```

has it's limits
you have to know the size on compilation time
size cannot change

Memory Allocation

memory can be allocated statically (on stack) or dynamically (on heap)
you already know static allocation

```
int i = 5;
double d = 6.2;
int a[5] = {1, 2, 3, 4, 5};
```

has it's limits
you have to know the size on compilation time
size cannot change
you can't create new objects

Memory Allocation

memory can be allocated statically (on stack) or dynamically (on heap)
you already know static allocation

```
int i = 5;
double d = 6.2;
int a[5] = {1, 2, 3, 4, 5};
```

has it's limits
you have to know the size on compilation time
size cannot change
you can't create new objects
it's all allocated on program start

Dynamic Allocation

Dynamic Allocation

much more flexible

Dynamic Allocation

much more flexible
danger of memory leak

Dynamic Allocation

much more flexible
danger of memory leak
need for manual deallocation

Dynamic Allocation

much more flexible
danger of memory leak
need for manual deallocation
you always get pointers (or objects)

Dynamic Allocation

much more flexible
danger of memory leak
need for manual deallocation
you always get pointers (or objects)
operators new and delete

Dynamic Allocation

much more flexible
danger of memory leak
need for manual deallocation
you always get pointers (or objects)
operators new and delete

```
int *i = new int(5);
delete i;
int *a = new int[5]{1, 2, 3, 4, 5};
delete[] i;
```

Dynamic Allocation


```
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    int sizeX = strtol(argv[1], NULL, 10);
    int sizeY = strtol(argv[2], NULL, 10);
    int **a = new int*[sizeX];
    for (int i = 0; i < sizeX; i++) {
        a[i] = new int[sizeY];
        for (int j = 0; j < sizeY; j++) {
            a[i][j] = i * sizeY + j;
            std::cout << a[i][j] << std::endl;
        }
    }
    for (int i = 0; i < sizeY; i++) {
        delete[] a[i];
    }
    delete[] a;
    return 0;
}
```

Questions?

Break!