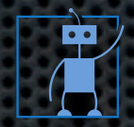


Robotics and
Embedded Systems

A whirlwind tour of C++

Echtzeitsysteme WS 2012/2013

heise@in.tum.de

What you should already know

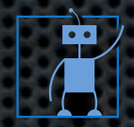
Robotics and
Embedded Systems

What you should already know

Robotics and
Embedded Systems

- The basic datatypes (e.g. int, float)

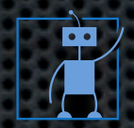
What you should already know



Robotics and
Embedded Systems

- The basic datatypes (e.g. int, float)
- Basic control flow (e.g. if/else, for, while)

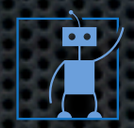
What you should already know



Robotics and
Embedded Systems

- The basic datatypes (e.g. int, float)
- Basic control flow (e.g. if/else, for, while)
- What methods and functions are

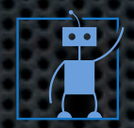
What you should already know



Robotics and
Embedded Systems

- The basic datatypes (e.g. int, float)
- Basic control flow (e.g. if/else, for, while)
- What methods and functions are
- What classes and objects are

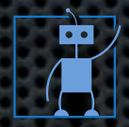
What you should already know



Robotics and
Embedded Systems

- The basic datatypes (e.g. int, float)
- Basic control flow (e.g. if/else, for, while)
- What methods and functions are
- What classes and objects are
- How to use a compiler

Hello C++

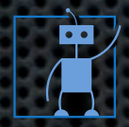


Robotics and
Embedded Systems

Code

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

Hello C++



Robotics and
Embedded Systems

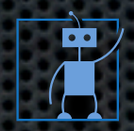
Code

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

Build/Output

```
$ g++ main.cpp
$ ./a.out
Hello World
$
```

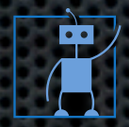
Functions



Robotics and
Embedded Systems

- Reuse and structure code
- Parameters and return value
- C++ allows pass by reference and value
- C++ allows function overloading

Functions

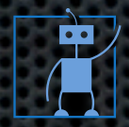


Robotics and
Embedded Systems

Code

```
#include <iostream>

int fac( int x )
{
    return ( x <= 1 ) ? 1 : x * fac( x - 1 );
}

int main()
{
    std::cout << fac( 5 ) << std::endl;
    return 0;
}
```

Functions



Robotics and
Embedded Systems

Code

```
#include <iostream>

int fac( int x )
{
    return ( x <= 1 ) ? 1 : x * fac( x - 1 );
}

int main()
{
    std::cout << fac( 5 ) << std::endl;
    return 0;
}
```

Output

```
$./a.out
120
$
```

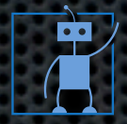
Functions

Robotics and Embedded Systems

- Return type
- Function name
- Argument 0 type
- Argument 0 name

```
int fac( int x )
{
    return ( x <= 1 ) ? 1 : x * fac( x - 1 );
}
```

Functions



Robotics and
Embedded Systems

- Return type
  - Function name
    - Argument 0 type
      - Argument 0 name

```
int fac( int x )
{
    return ( x <= 1 ) ? 1 : x * fac( x - 1 );
}
```

arbitrary number of arguments possible

```
type function( type0 arg0, type1 arg1, ..., typeN argN )
{
   ...
}
```

Functions

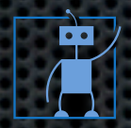
Pass by value vs. reference

Robotics and Embedded Systems

```cpp
#include <iostream>

void func_value( int x )
{
    x = 10;
}

void func_reference( int& x )
{
    x = 10;
}

int main()
{
    int a = 0;
    func_value( a );
    std::cout << a << std::endl;
    func_reference( a );
    std::cout << a << std::endl;
    return 0;
}
```

Functions

Pass by value vs. reference



Robotics and
Embedded Systems

```cpp
#include <iostream>

void func_value( int x )
{
    x = 10;
}

void func_reference( int& x )
{
    x = 10;
}

int main()
{
    int a = 0;
    func_value( a );
    std::cout << a << std::endl;
    func_reference( a );
    std::cout << a << std::endl;
    return 0;
}
```

Output

```
$./a.out
0
10
$
```

Functions

Overloading

Robotics and Embedded Systems

```cpp
#include <iostream>

void func( int v )
{
    std::cout << "Integer: " << v << std::endl;
}

void func( float v )
{
    std::cout << "Float: " << v << std::endl;
}

int main()
{
    func( 5 );
    func( 1.0f );
    return 0;
}
```

Functions

Overloading

Robotics and Embedded Systems

```cpp
#include <iostream>

void func( int v )
{
    std::cout << "Integer: " << v << std::endl;
}

void func( float v )
{
    std::cout << "Float: " << v << std::endl;
}

int main()
{
    func( 5 );
    func( 1.0f );
    return 0;
}
```

Output

```
$./a.out
Integer: 5
Float: 1
$
```

Arrays

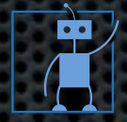
Robotics and Embedded Systems

Declaration

Type
Name
Dimension

```
type name[ dimension ];
type name[ dimension1 ][ dimension2 ];
...
```

Arrays



Robotics and
Embedded Systems

Declaration

Type
Name
Dimension

```
type name[ dimension ];
type name[ dimension1 ][ dimension2 ];
...
```

Initialization

```
int array[ 4 ];
array[ 0 ] = 0;
array[ 1 ] = 5;
array[ 2 ] = 8;
array[ 3 ] = 3;
```

```
int array[ 4 ] = { 3, 7, 9, 2 };
```

```
int array[] = { 3, 7, 9, 2 };
```

Arrays

Robotics and Embedded Systems

- Special initialization for char arrays / strings
- The following char arrays are equivalent

```
char str[] = "String";
char str2[] = { 'S','t','r','i','n','g','\0' };
```

Pointers

Robotics and
Embedded Systems

Pointers

Robotics and
Embedded Systems

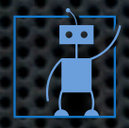
- A variable name refers to a particular location in memory and stores a value there

Pointers

Robotics and
Embedded Systems

- A variable name refers to a particular location in memory and stores a value there
- If you refer to the variable by name then

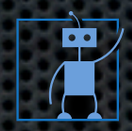
Pointers



Robotics and
Embedded Systems

- A variable name refers to a particular location in memory and stores a value there
- If you refer to the variable by name then
  - the memory address is looked up

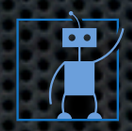
Pointers



Robotics and
Embedded Systems

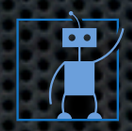
- A variable name refers to a particular location in memory and stores a value there
- If you refer to the variable by name then
  - the memory address is looked up
  - the value at the address is retrieved or set

Pointers



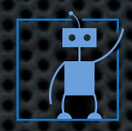
- A variable name refers to a particular location in memory and stores a value there
- If you refer to the variable by name then
  - the memory address is looked up
  - the value at the address is retrieved or set
- C++ allows us to perform these steps independently

Pointers



- A variable name refers to a particular location in memory and stores a value there
- If you refer to the variable by name then
  - the memory address is looked up
  - the value at the address is retrieved or set
- C++ allows us to perform these steps independently
  - `&x` evaluates to the address of `x` in memory

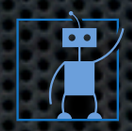
Pointers



Robotics and
Embedded Systems

- A variable name refers to a particular location in memory and stores a value there
- If you refer to the variable by name then
  - the memory address is looked up
  - the value at the address is retrieved or set
- C++ allows us to perform these steps independently
  - `&x` evaluates to the address of `x` in memory
  - `*( &x )` dereferences the address of `x` and retrieves the value of `x`

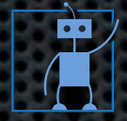
Pointers



- A variable name refers to a particular location in memory and stores a value there
- If you refer to the variable by name then
  - the memory address is looked up
  - the value at the address is retrieved or set
- C++ allows us to perform these steps independently
  - `&x` evaluates to the address of x in memory
  - `*( &x )` dereferences the address of x and retrieves the value of x
  - `*( &x )` is the same thing as x

Pointers

Code



Robotics and
Embedded Systems

```cpp
#include <iostream>

int main()
{
    int x;
    int* p = &x;

    x = 10;
    std::cout << *p << std::endl;

    *p = 5;
    std::cout << x << std::endl;

    return 0;
}
```

Pointers

Code

```cpp
#include <iostream>

int main()
{
    int x;
    int* p = &x;

    x = 10;
    std::cout << *p << std::endl;

    *p = 5;
    std::cout << x << std::endl;

    return 0;
}
```

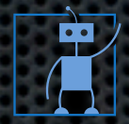
Output

```
$./a.out
10
5
$
```

Pointers

Robotics and Embedded Systems

Pointers



Robotics and
Embedded Systems

- Example

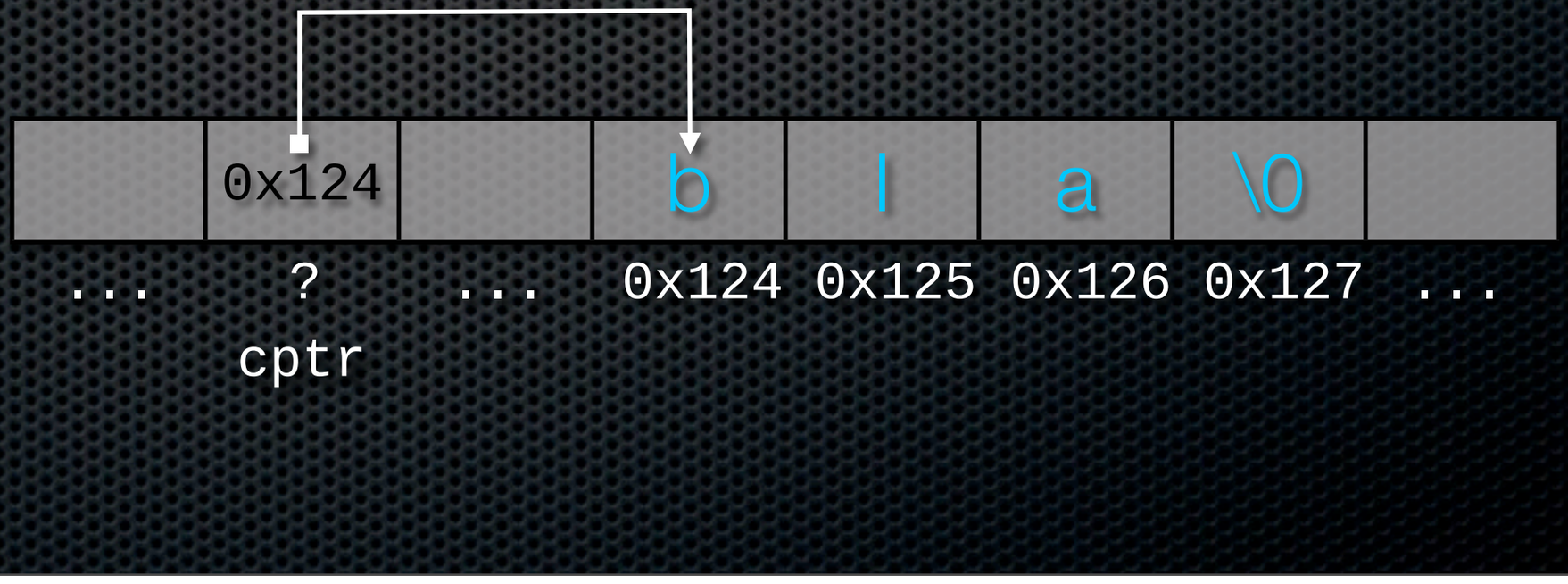
```
#include <iostream>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }
    std::cout << len << std::endl;
}
```

Pointers

Robotics and Embedded Systems

- Example

```
#include <iostream>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }
    std::cout << len << std::endl;
}
```

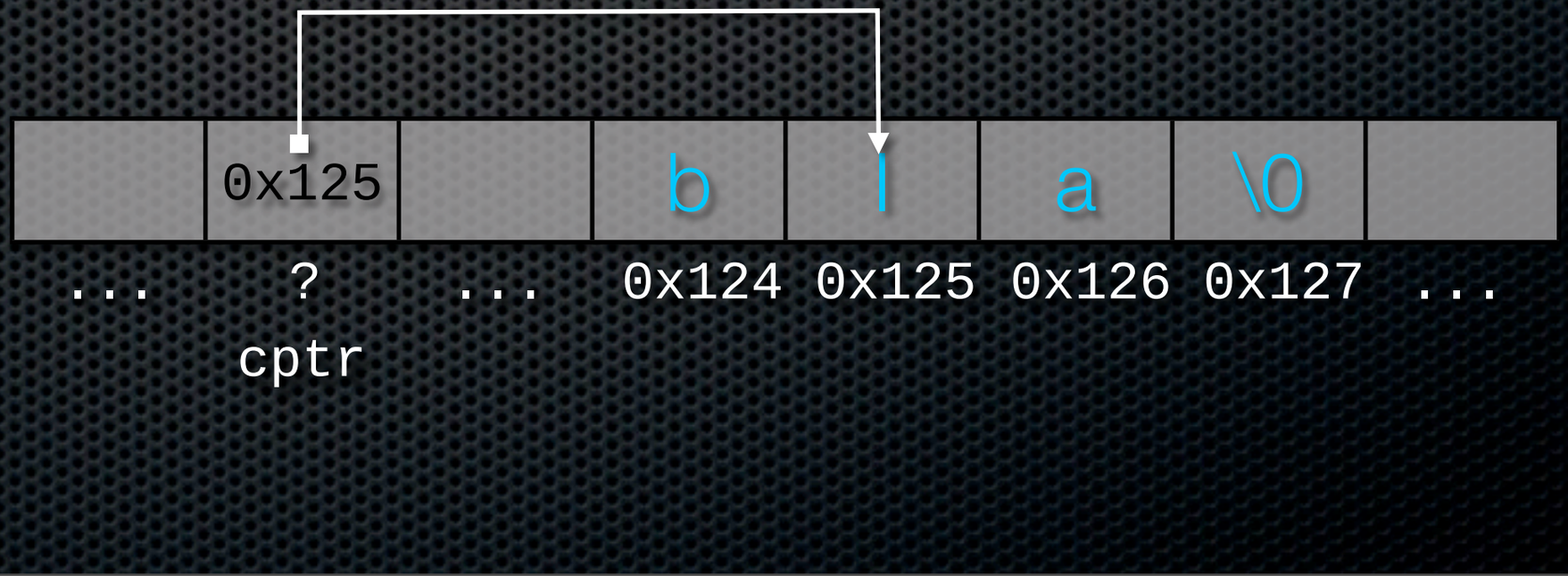


Pointers

Robotics and Embedded Systems

- Example

```
#include <iostream>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }
    std::cout << len << std::endl;
}
```

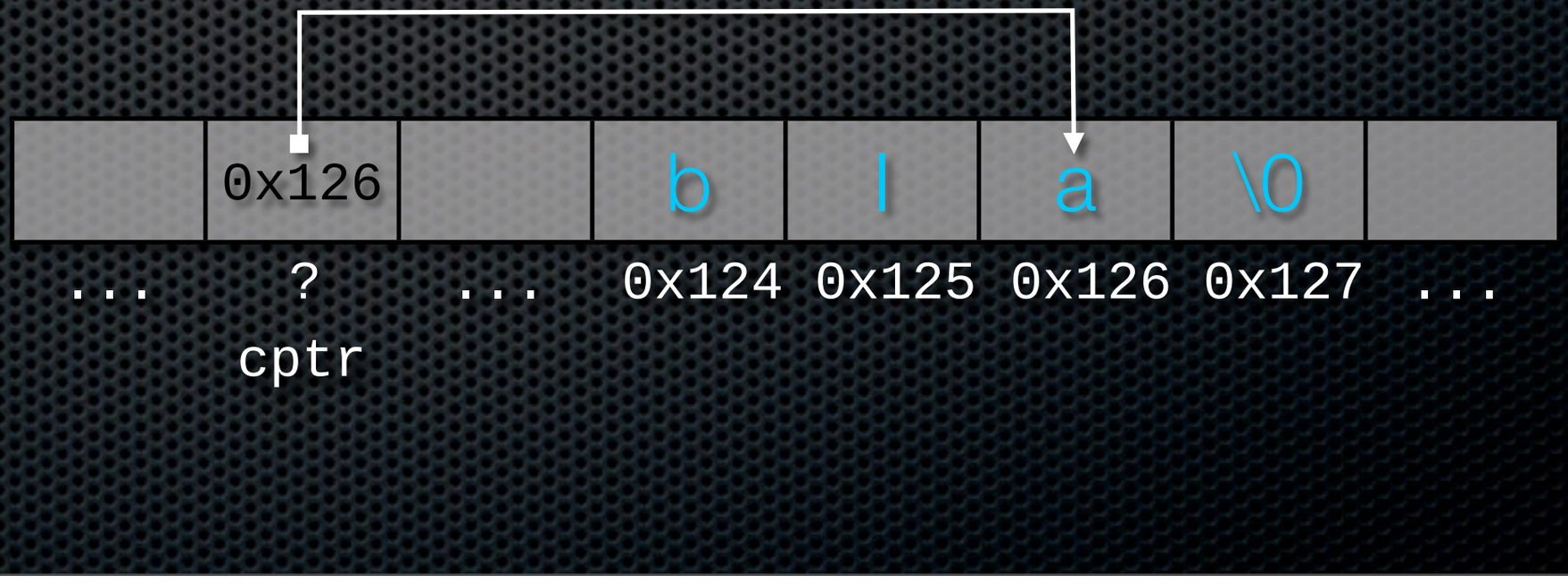


Pointers

Robotics and Embedded Systems

- Example

```
#include <iostream>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }
    std::cout << len << std::endl;
}
```

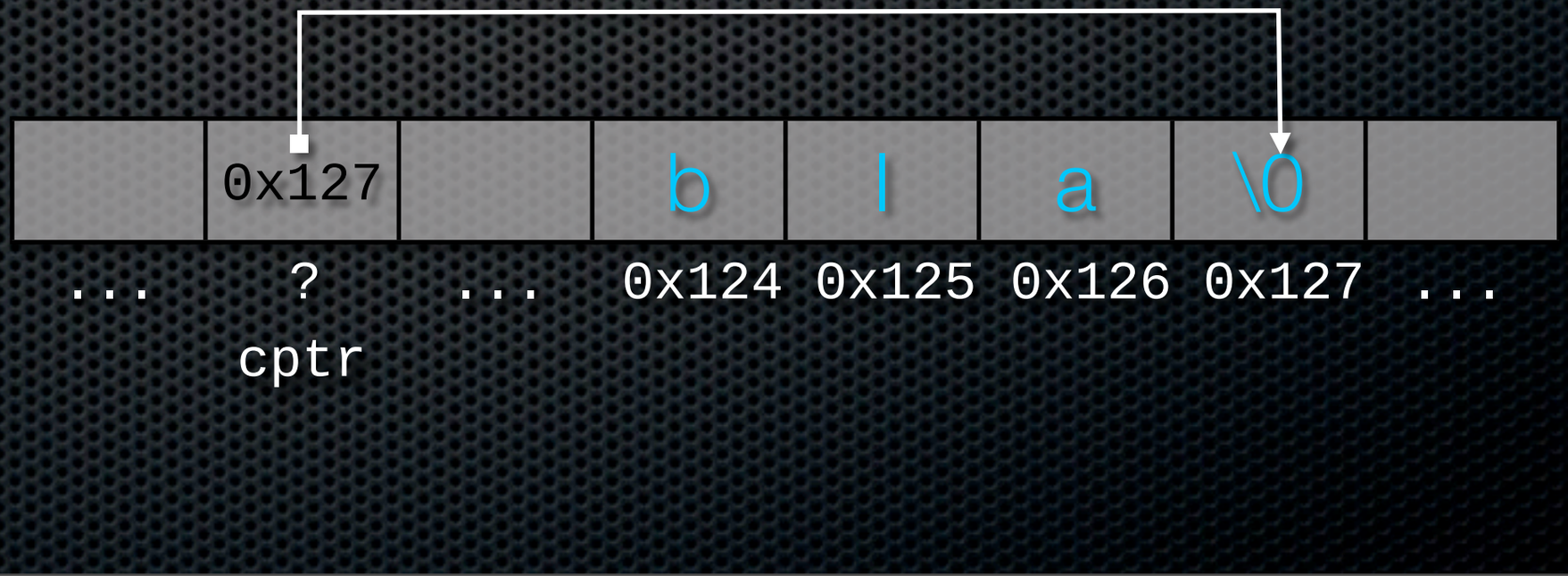


Pointers

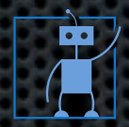
Robotics and Embedded Systems

- Example

```
#include <iostream>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }
    std::cout << len << std::endl;
}
```



Pointers



Robotics and
Embedded Systems

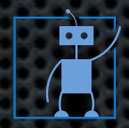
- Pointers and arrays

```
int array[ 5 ];
...
         array ≡ &array[ 0 ]
        *array ≡  array[ 0 ]
*( array + 1 ) ≡  array[ 1 ] ≡ 1[ array ]
...
```

- Arithmetic pointer operations modify the address by sizeof(type) bytes

```
#include <iostream>

int main()
{
    char*  x = 0x0;
    float* y = 0x0;

    std::cout << ( void* ) ( x + 1 ) << std::endl;
    std::cout << ( void* ) ( y + 1 ) << std::endl;
}
```

Pointers



Robotics and
Embedded Systems

- Pointers and arrays

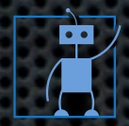
```
int array[ 5 ];
...
         array ≡ &array[ 0 ]
        *array ≡  array[ 0 ]
*( array + 1 ) ≡  array[ 1 ] ≡ 1[ array ]
...
```

- Arithmetic pointer operations modify the address by sizeof(type) bytes

```
#include <iostream>

int main()
{
    char*  x = 0x0;
    float* y = 0x0;

    std::cout << ( void* ) ( x + 1 ) << std::endl;
    std::cout << ( void* ) ( y + 1 ) << std::endl;
}
```

```
$ ./a.out
0x1
0x4
$
```

Pointers



Robotics and
Embedded Systems

```
const int* ptr
```

```
int* const ptr
```

```
const int* const ptr
```

Pointers

Robotics and Embedded Systems

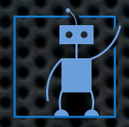
```
const int* ptr
```

- Declares a changeable pointer to a constant integer
- value cannot be changed
- pointer can be changed to point to a different constant integer

```
int* const ptr
```

```
const int* const ptr
```

Pointers



Robotics and
Embedded Systems

```
const int* ptr
```

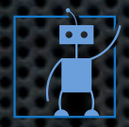
- Declares a changeable pointer to a constant integer
- value cannot be changed
- pointer can be changed to point to a different constant integer

```
int* const ptr
```

- Declares a constant pointer to a changeable integer
- value can be changed
- pointer cannot be changed to point to a different integer

```
const int* const ptr
```

Pointers



Robotics and
Embedded Systems

```
const int* ptr
```

- Declares a changeable pointer to a constant integer
- value cannot be changed
- pointer can be changed to point to a different constant integer

```
int* const ptr
```

- Declares a constant pointer to a changeable integer
- value can be changed
- pointer cannot be changed to point to a different integer

```
const int* const ptr
```

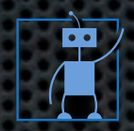
- Neither the value nor the address can be changed

Pointers

- No guarantees that a pointer points to a valid address

```
...

int* ptr = 0xdeadbeef;
int* ptr = 0x0;

...

int* function()
{
    int x;
    return &x;
}

...

int* p = new int[ 5 ];
delete p;

...
```

Memory management



Robotics and
Embedded Systems

- Dynamic memory allocation possible using new/delete

```
...
int* x = new int;
...
int* y = new int[ 10 ];
...
float** z;
z = new float*[ 2 ];
z[ 0 ] = new float[ 3 ];
z[ 1 ] = new float[ 4 ];
...
delete x;
delete[] y;
delete[] z[ 0 ];
delete[] z[ 1 ];
delete[] z;
...
```

- If allocated memory is not correctly freed using delete it is wasted and cannot be reused
- Pointers to deleted memory still contain the address

Classes

Robotics and Embedded Systems

- Make the coupling between functions and data explicit
- Allows the definition of new datatypes
- Enhanced reusability and readability

Classes

Robotics and Embedded Systems

Visibility

Class name

```
class name
{
    public:
        ... methods/members ...

    private:
        ... methods/members ...

    protected:
        ... methods/members ...
};
```

Classes

Robotics and Embedded Systems

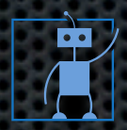
Visibility

Class name

```
class name
{
    public:
        ... methods/members ...

    private:
        ... methods/members ...

    protected:
        ... methods/members ...
};
```

- Public members/methods can be accessed from outside

Classes



Robotics and
Embedded Systems

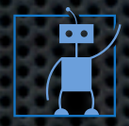
Visibility

Class name

```
class name
{
    public:
        ... methods/members ...

    private:
        ... methods/members ...

    protected:
        ... methods/members ...
};
```

- Public members/methods can be accessed from outside
- Private/protected members/methods can only be accessed from within the class

Classes



Robotics and
Embedded Systems

```cpp
#include <iostream>

class Complex
{
    public:
        Complex( float r, float i ) { re = r; im = i; }

        void print() { std::cout << "( " << re << " , " << im << " )" << std::endl; }

        float re;
        float im;
};

int main()
{
    Complex c( 1.0f, 0.0f );
    c.print();
    c.re = 2.0f;
    c.print();
}
```

Classes

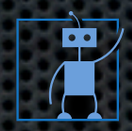
Robotics and Embedded Systems

```cpp
#include <iostream>

class Complex
{
    public:
        Complex( float r, float i ) { re = r; im = i; }

        void print() { std::cout << "( " << re << " , " << im << " )" << std::endl; }

        float re;
        float im;
};

int main()
{
    Complex c( 1.0f, 0.0f );
    c.print();
    c.re = 2.0f;
    c.print();
}
```

Output

```
$./a.out
( 1 , 0 )
( 2 , 0 )
$
```

Classes



Robotics and
Embedded Systems

- Special methods for construction and deconstruction (constructor/destructor)

```cpp
#include <iostream>

class Foobar
{
    public:
        Foobar()  { std::cout << "ctor" << std::endl; }
        ~Foobar() { std::cout << "dtor" << std::endl; }
};

int main()
{
    Foobar obj;
}
```

Classes

- Special methods for construction and deconstruction (constructor/destructor)

```cpp
#include <iostream>

class Foobar
{
    public:
        Foobar()  { std::cout << "ctor" << std::endl; }
        ~Foobar() { std::cout << "dtor" << std::endl; }
};

int main()
{
    Foobar obj;
}
```

Output

```
$./a.out
ctor
dtor
$
```

Classes

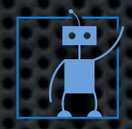
Robotics and
Embedded Systems

Classes

Robotics and
Embedded Systems

- Constructor brings the object into a consistent state

Classes



Robotics and
Embedded Systems

- Constructor brings the object into a consistent state
- Deconstructor can be used for cleaning up (especially useful for dynamic memory)

Classes

Robotics and Embedded Systems

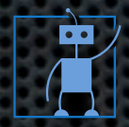
- Constructor brings the object into a consistent state
- Deconstructor can be used for cleaning up (especially useful for dynamic memory)
- More special methods exist e.g. for copying objects and special operators

Classes

- If pointers to objects are used, then methods/members can be accessed via “->”

```
#include <iostream>

class Blub
{
    public:
        Blub( int x ) { bla = x; }
        int bla;
};

int main()
{
    Blub* x = new Blub( 2 );

    std::cout << ( *x ).bla << std::endl;
    std::cout << x->bla << std::endl;
}
```

Classes



Robotics and
Embedded Systems

- If pointers to objects are used, then methods/members can be accessed via “->”

```
#include <iostream>

class Blub
{
    public:
        Blub( int x ) { bla = x; }
        int bla;
};

int main()
{
    Blub* x = new Blub( 2 );

    std::cout << ( *x ).bla << std::endl;
    std::cout << x->bla << std::endl;
}
```

Output

```
$./a.out
2
2
$
```

Robotics and
Embedded Systems

Questions?