Part 2: Datatypes, Pointer and References, Basic Arrays

Prof. Dr. Ulrik Schroeder

C++ - Einführung ins Programmieren

WS 03/04

http://learntech.rwth-aachen.de/lehre/ws04/c++/folien/index.html

Basic Types in C++ (and strings)

- not in all details
  - just the differences to Java

- Integer values
  - counting, indexing, real world integer values, models
- Real numbers
  - floating point (not continuous !!, limits in precision)
  - physical values
- Characters
  - human readable decodification of data
  - (strings)
- Boolean
  - conditions

basic (built-in) types

- main difference: missing standards (size, sequence of bytes)
  - only ANSI C++ definition
    - char <= short <= int <= long

sizeof(<<name>>) operator

- symbolic constants for implementation dependent values

  <climits>

  <cfloat>

INT_MIN, INT_MAX, SHRT_MIN.; ...

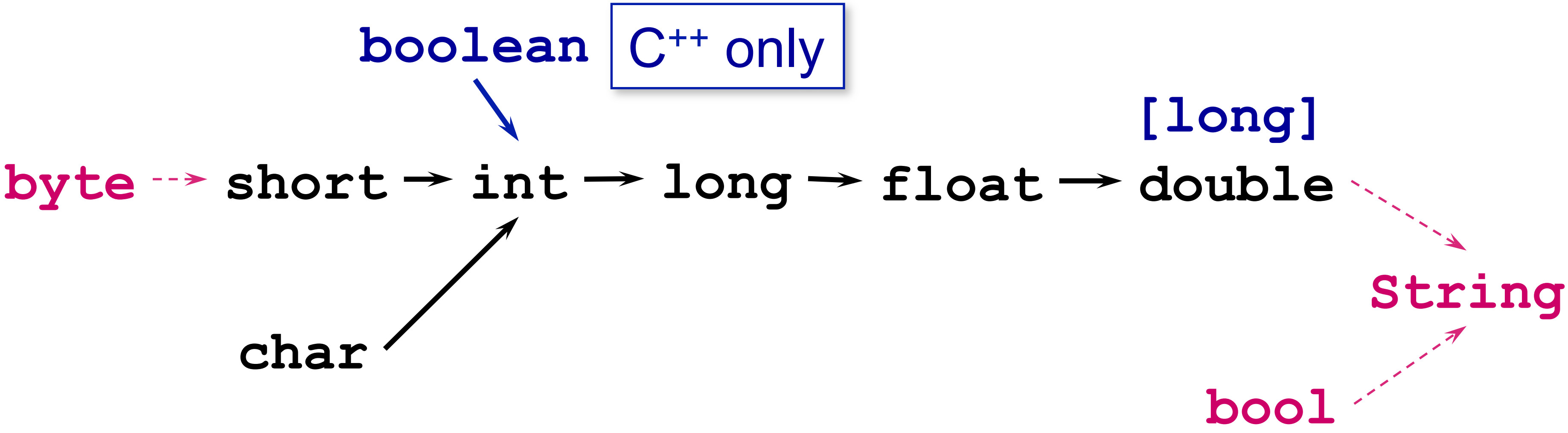
- bool (compatible with integer):
  - 0 = false, everything else (interpreted as 1) = true

- char (ASCII), wchar_t (wide character for unicode)

- int, short [int], long [int]

- each (arithmetic, char) type can be signed or unsigned

implicit conversions



Java only

- C++ has no implicit conversion to String (for output reasons)
- << operator is heavily overloaded to compensate for this, but must also be defined for user defined types (all your classes!)
- explicit type *casts* like Java:
  - (int) 3.14159 == 3

String literals

- Strings are no built in type, but have literal representations

```
"this is a string constant"
```

- Strings are equivalent to arrays of char (ending with '\0')

'r'	'w'	't'	'h'	'\0'

```
char a[ 5 ] = "rwth";
```

a

```
sizeof( "rwth" ) == 5;    a[ 0 ] == 'r'    a[ 4 ] == '\0'
```

- arrays of char are equivalent to pointer (later topic)

```
char a[ 5 ] = "rwth";
char* b = a;
cout << b;          → rwth
b[ 1 ] = 'u';
cout << a;          → ruth
```

C++ also has class string (comparable to Java String)

class string

- #include <string>
- Create:

```
string    s = "copy of this String literal.",
          t( s ),
          u( "literal" ),
          v( '=', 40 );
```

without '\0' at the end!!

copy constructor

constructor(char[ ])

constructor(char c, int anz)

"=="

- manipulate
  - concat – Ops: + and `s += " appended"` (like java)
  - compare: `s1 == s2` (Java: `s1.equals( s2 )`)
  - find substring: `int pos = s.find( "this" ); if ( pos != string::npos ) ...`
  - replace: `s.replace( pos, 4, "another" );`

string manipulation

```
#include <iostream> .. <string> .. <ctime>
using namespace std;
int main( ) {
    time_t sek;
    time( &sek );              // get date & time (long)
    string t = ctime( &sek ); // convert to strin
    string dayOfWeek( t, 0, 3 ),
           month( t, 4, 3 ),
           day( t, 8, 2 ),
           year( t, t.size( ) - 5, 4 );
    cout << dayOfWeek+" " << day+". " << month+". "
         << year+"\n";
} // main( )
```

time_t, time( ), ctime( )

class string ...

1041015724

Fri Dec 27 19:27:45 2002

new substring from start, 3 chars long

?

Fri 27. Dec. 2002

declaration of objects

- local variables, global variables, parameters, attributes

```
const double  pi = 3.141593;
volatile clock_t ticks;
register int counter;

double pow( double base, double exponent ); // better prototype
int rand( void );        // same as int rand( )

string s( "create string object" );
point u, v( 2, 3 );
```

must be initialized here
can not be changed

changed outside of program

heavily used optimization??

random: 0 .. 32767

constructor without new X( )

C++ pattern: Use const as much as possible (arguments, …)

Java -> C++: basics

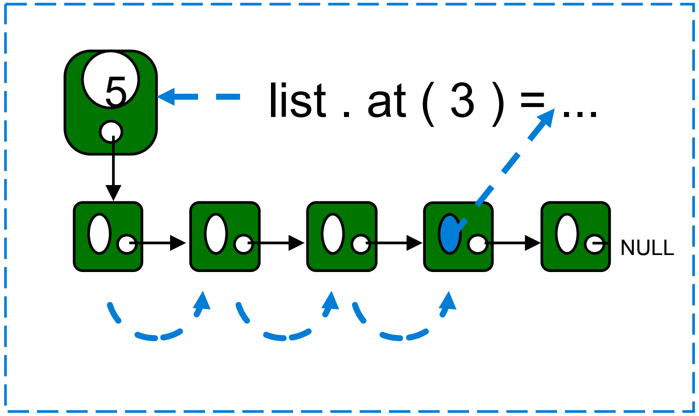
- C^{++} / Java basic differences & philosophy
- Separation of declaration & definition (*.h & *.cpp)
  - edit / compile / link / test cycle, makefiles
- IO
  - streams, file streams, manipulators
- namespaces
  - std for cout
- basic types

- expressions & statements are practically the same in Java and C^{++}
  - (but sequence of evaluation in expressions not standardized)

```
int i = 2; cout << i-- * i++; // g++: 2, CC: 4
```

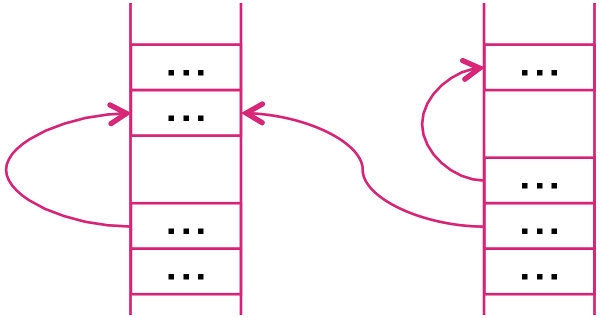
Pointer & References

- references of objects
  - call by reference



- pointers
  - usage patterns
  - pointers & arrays

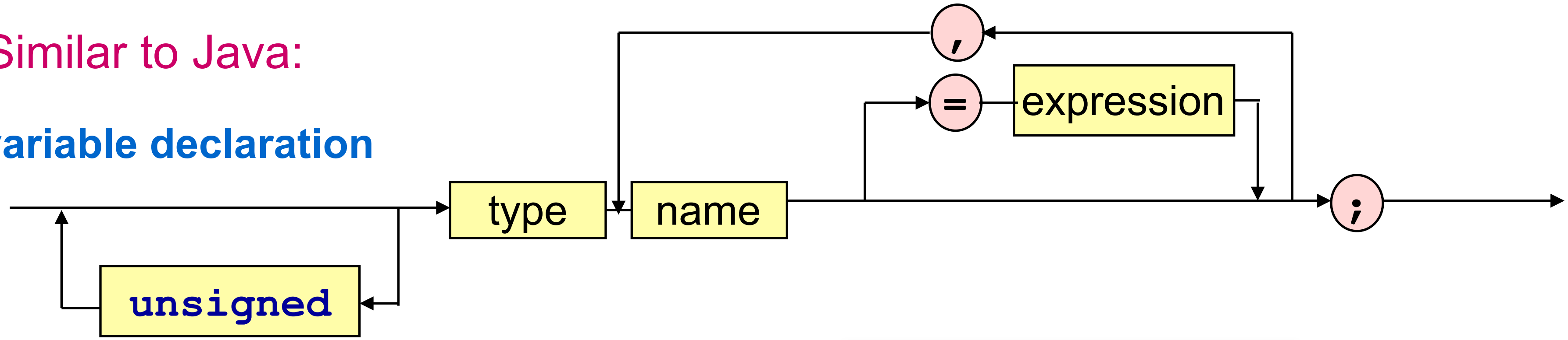
- storage classes
  - local objects => auto (stack)
  - global objects => extern (heap)
  - objects "pointed to" & static objects



Declaration of variables

Similar to Java:

variable declaration




```
          short int i = 256S;
                int j = 42, k;
unsigned long   int l = 123456uL;
float               x = 42.1;
double              y = 123.456d;
bool                b = true;
```

numeric basic type names and literals.

References: T &

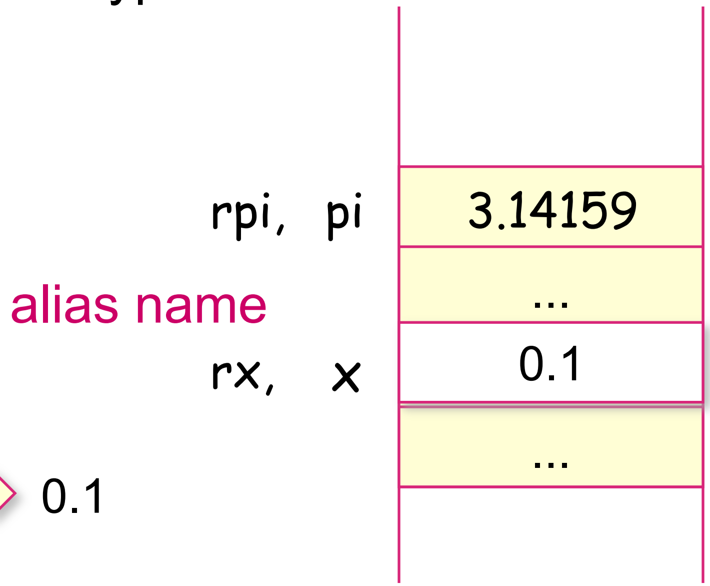
- reference = alias name for an existing object
- syntax:
  - declaration: Typename& varname;
    - "varname is reference to variable of Typename"

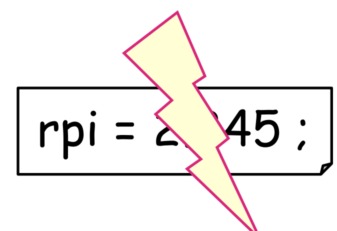
explicit reference semantics as Java (implicit) class types

```
...
float x = 42.1f;
...
float& rx = x;
rx /=  421 . 0 ;
cout << "x = " << x;
const double pi = 3.14159 ;
const double & rpi = pi ;
```

→ 0.1

read only reference!



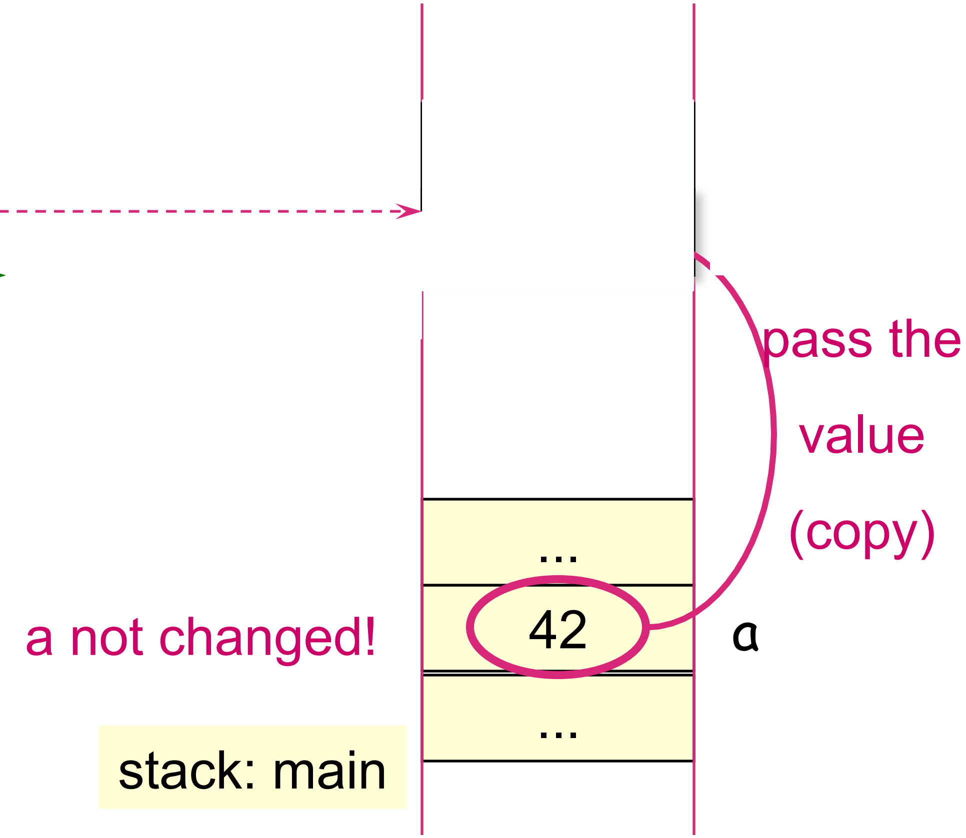


Use case for references

- reference declarations are used in C++ to implement **call by reference** parameter passing
- normal declaration: **call by value**
  - value of argument (expression) is copied into local parameter

```
const int STEP = 5;
void nextStep( int var ) {
    var += STEP;
} // nextStep( )

int main( ) { ...
    int a = 42;
    nextStep( a );
    cout << "a=" << a << endl;
} // main( )
```

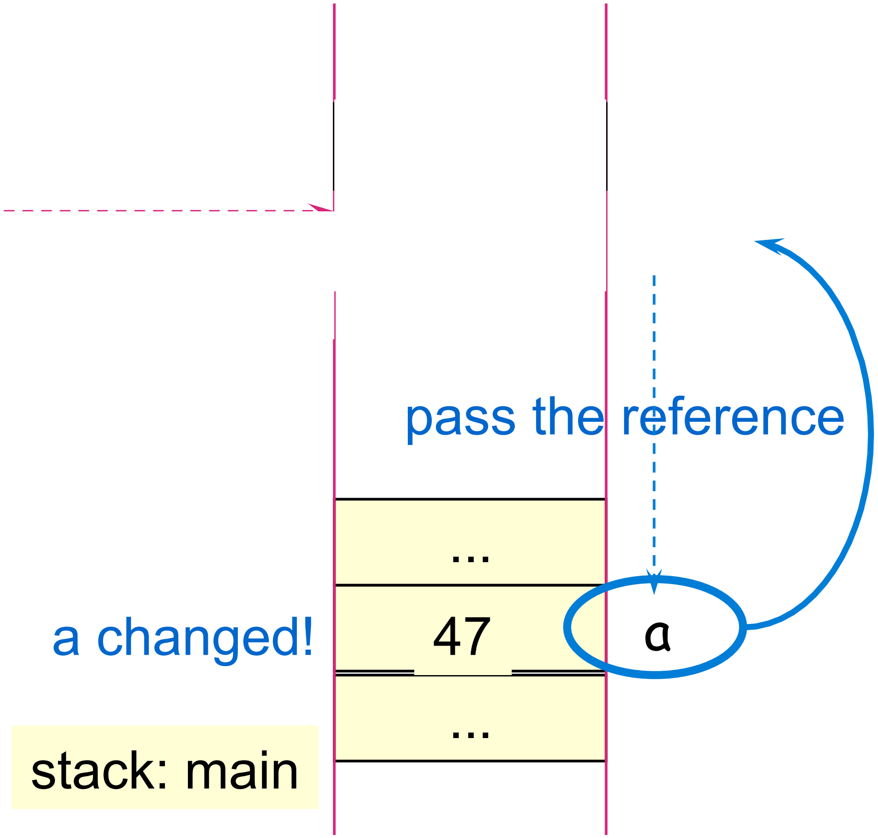


Use case for references

- reference declarations are used in C++ to implement call by reference parameter passing: call by reference
    - reference to the argument is passed to the formal parameter

```
const int STEP = 5;
void nextStep( int& var ) {
    var += STEP;
} // nextStep( )

int main( ) { ...
    int a = 42;
    nextStep( a );
    cout << "a=" << a << endl;
} // main( )
```

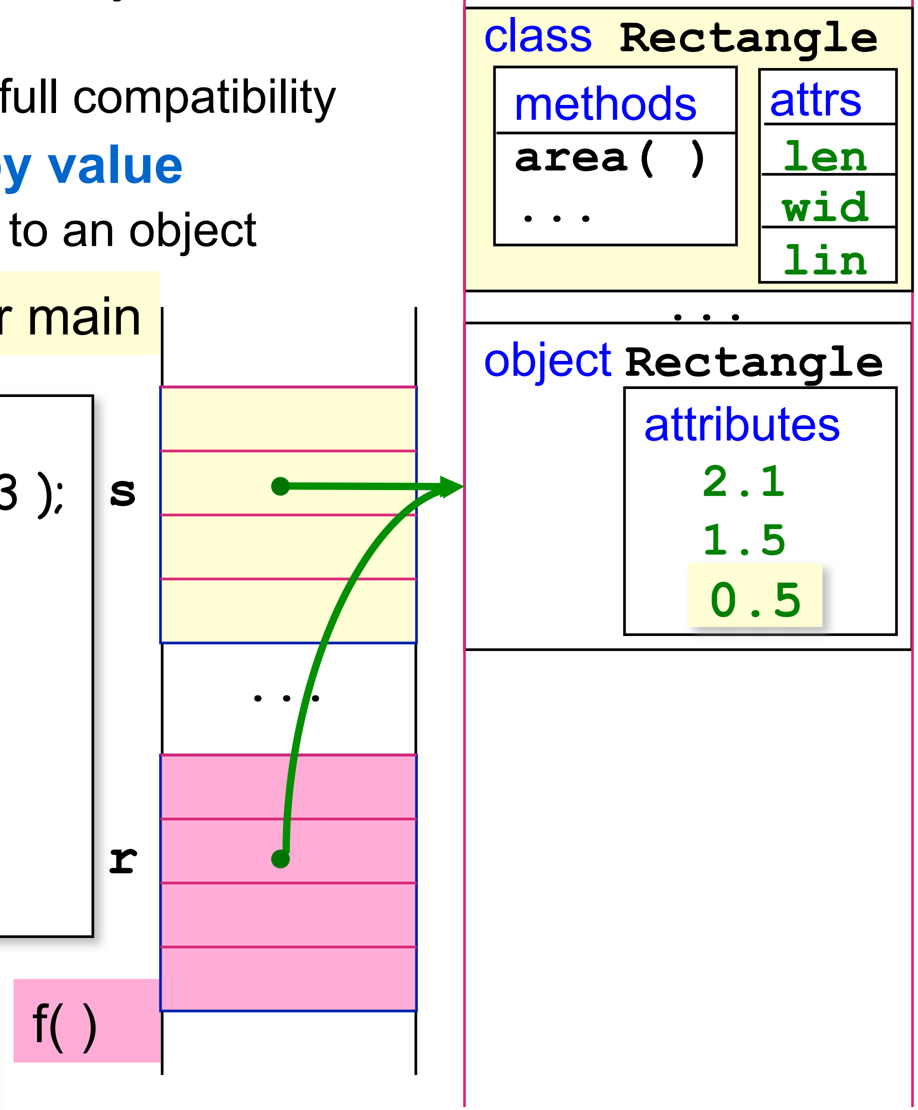


Reminder: java reference semantics

- **Java** has **reference semantics** for objects and **value semantics** for basic (built-in) types
  - wrapper classes for basic types for full compatibility
- the only parameter concept: **call by value**
  - … but the value can be a reference to an object

```
public static void main (...) {
    Rectangle s = new Rectangle( 2.1, 1.5, 3 );
    f( s );
} // main( )

public static void f ( Rectangle r ) {
    r.setLineStrength( 0.5 );
    r = new Rectangle ( 2.5, 2.0, 1 );
} // f( )
```



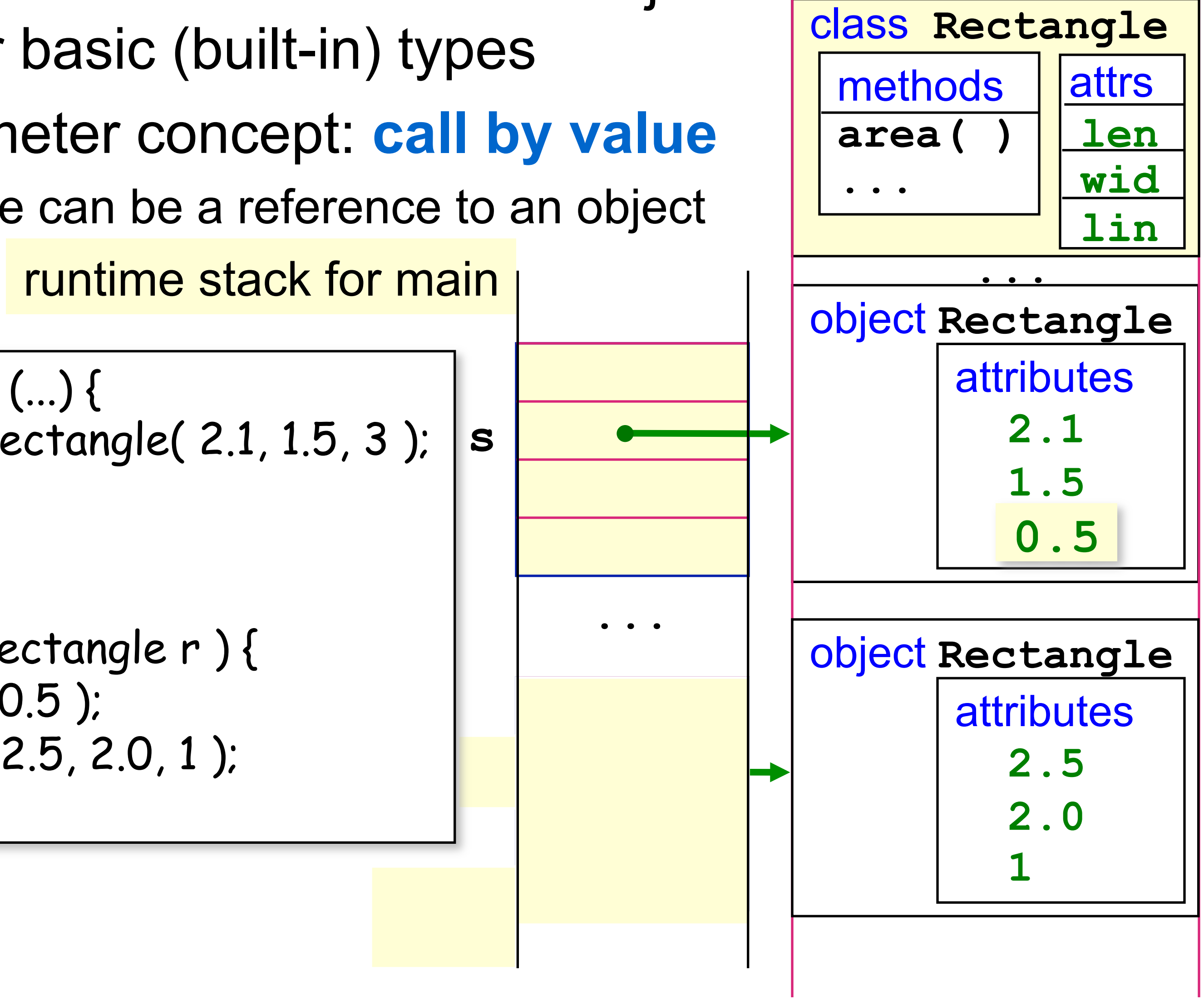
f( ) can not change the object (the reference), but its state !

reminder: java reference semantics

- **Java** has **reference semantics** for objects and **value semantics** for basic (built-in) types
- the only parameter concept: **call by value**
  - … but the value can be a reference to an object

```
public static void main (...) {
    Rectangle s = new Rectangle( 2.1, 1.5, 3 );
    f( s );
} // main( )

public static void f ( Rectangle r ) {
    r.setLineStrength( 0.5 );
    r = new Rectangle ( 2.5, 2.0, 1 );
} // f( )
```

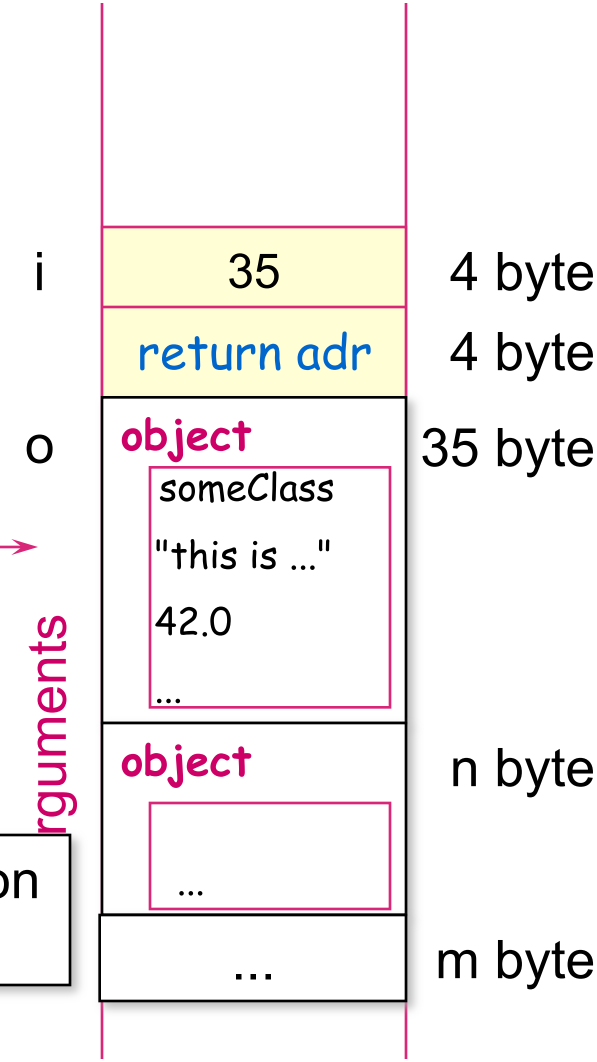


Call by value vs. reference (C++)

- size of arguments with large objects => better runtime

```
void display(  someClass    o, ...  ) {
    int i = sizeof( o ); ...
    cout << s + i << endl;
} // display( )
int main( ) {
    someClass obj( ... ); ...
    display( obj, ... );
} // main( )
```

stack: function display



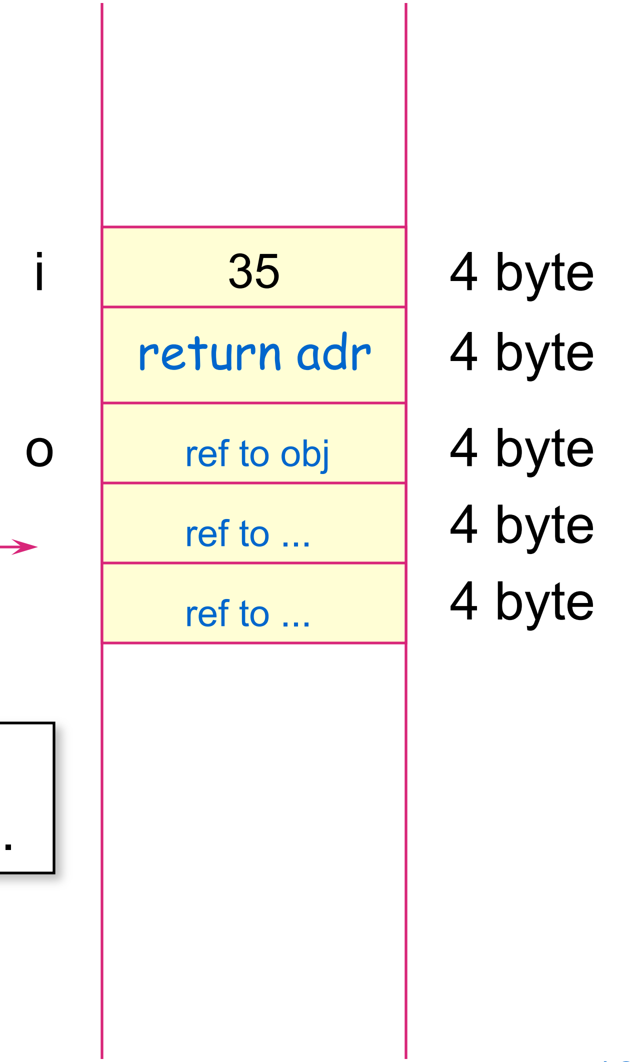
temporary objects created when calling the function and deleted when it is finished

Call by value vs. reference (C++)

- size of arguments with large objects => better runtime

```
void display(  someClass& o, ...  ) {
    int i = sizeof( o ); ...
    cout << s + i << endl;
} // display( )
int main( ) {
    someClass obj( ... ); ...
    display( obj, ... );
} // main( )
```

stack: function display



references are passed as addresses to the storage for which the alias name is defined.

Call by value vs. reference (C++)

- size of arguments with large objects => better runtime
- side effects: returning more than one result
  - !! controversial !!

```
void swap(  int& x, int& y  ) {
    int help = x;
    x = y; y = help;
} // swap( )

int main( ) {
    for ( ...
        if ( a[ i ] > a[ j ] )
            swap( a[ i ], a[ j ] );
} // main( )
```

can not be called with literal values:

swap(5, 7);

the values of a[i] and a[j] are swapped

References as return values

- if objects are returned => save runtime to pass the reference

```
string& msg( ) {
    static string s="this is the message";
    return s;
} // msg( )

int main( ) {
    string x = "!!!";
    msg( ) += x;
    cout << msg( );
} // main( )
```

can not return reference to a local object!

stored in **global store**, stays valid until end of program

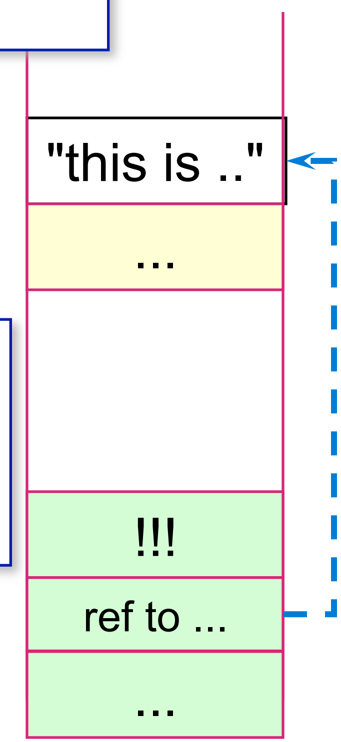
?

but: object is created only once!!

most cases need newly created objects per function call!

solution: Pointer

<<noname>>



References as return values

```
double& max(double &, double &);
int main( ) {
    double x = 1.7, y = 42.3;
     x += ++max( x, y );
    max( x, y ) += 5.0;
    cout << "x = " << x << ", y = " << y << endl;
} // main( )

double& max( double & a, double & b ) {
    return a > b ? a : b;
} // max( )
```

?

x	y	max(x, y)
1.7	42.3	
	++	ref to y
+=	43.3	
45.0		
	+=	ref to x
50.0		

x = 50, y = 43.3

References as return values

- many operators are functions returning references to objects

```
cout << "x = " << x << ", y = " << y << endl;
```

```
cout << "x =  "
```

reference to cout `<< x`

reference to cout `<< ", y = "`

reference to cout

```
ostream& operator << ( ostream& o, double x ) {
        o.put( x );
        return o;
} // operator <<
```

Operator << takes two arguments and returns the output stream it writes to.

Reference Semantics

- Java:
  - automatically for all objects (class types)

- C++
  - must be explicitly defined by the programmer
    - Type & name
  - practical uses
    - call by reference function arguments (large objects)
    - out-parameters to return more than one value (swap)
    - reference to objects to work within a pipeline (cout)

Pointer: T *

- an expression which evaluates to an address and a type
- declaration: Typename* variablename;

```
int    i = 42,
      * ip = & i;
int* jp; // no allocation!
```

variable	content (type)		address
...	...	...	...
i	42	(int)	0x415004
ip	0x415004	(* int)	0x415008
jp	0	(* int)	0x41500c
...	...	...	...

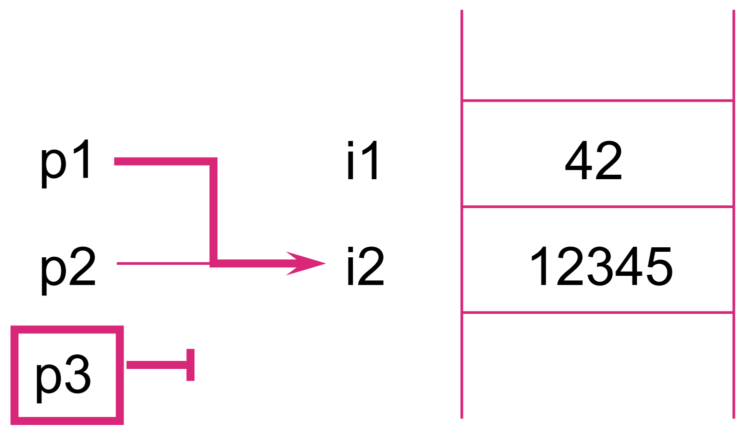
default value: NULL

Address-Operator &

Pointer variables and return values

- Pointers and references are somehow equivalent (but not the same!!!)
  - references are constant addresses of existing objects
  - these addresses can not be changed (by the programmer)
- Pointers are variables of addresses of (possible) objects
  - these variables can be changed (can point to different objects)

```
int i1 = 0, i2 = 100,
int *p1 = &i1, *p2 = &i2;
*p1 = 42;   // as i1=42
*p2 = *p1; // as i2=i1
p1 = p2;
*p1 = 12345;
int* p3;
```

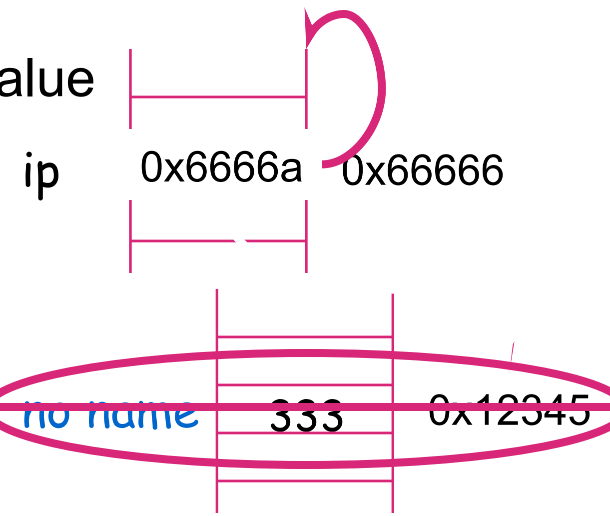


p3 == NULL   special value 0x00000000

Allocation and Deletion

```
int* ip  = new int( 333 );
. . .
delete ip;
ip = &otherInt;
```

allocate store & assign value



variable ip holds new address (points to another int)

leakage!

store stays allocated

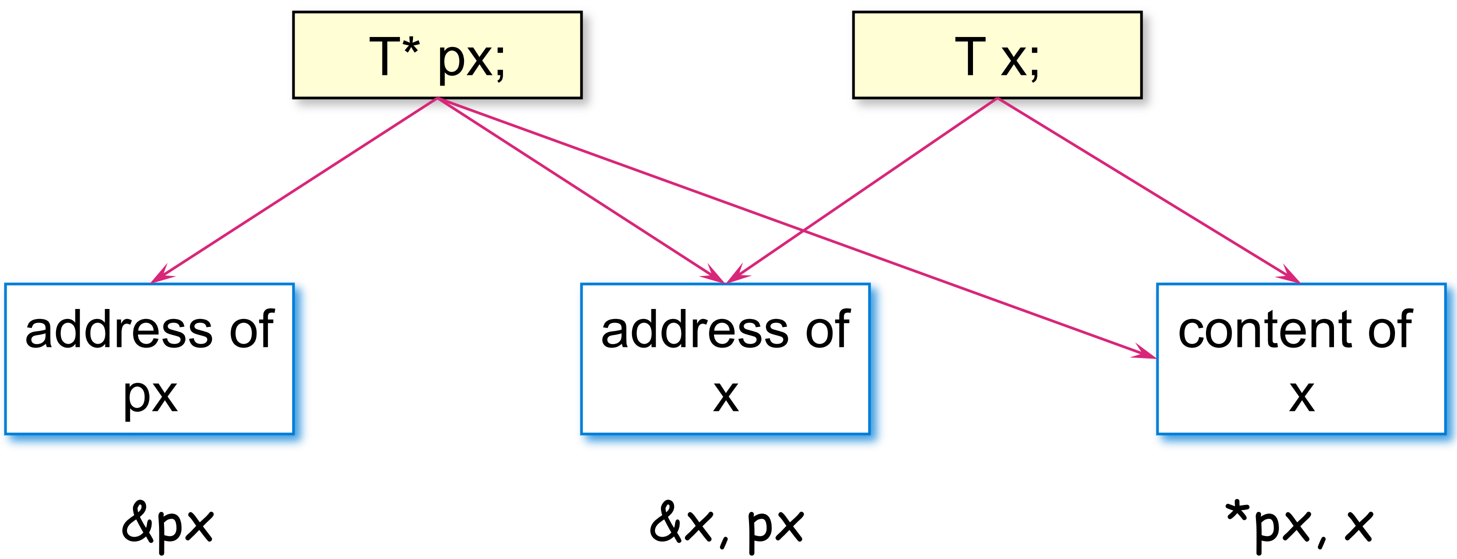
can not be reached

no automatic garbage collection

??? difference between int* ip and int *ip ; ???

Operators

- Dereferencing operator * (=> object pointed by)
- Reference-operator & (address of the object)
- declaration of a reference T&
- declaration of a pointer T*



T** ppx; pointer to a pointer to an object of type T

const and pointer

- values not to be changed: declared as `const`
- which of the following are legal?

```
int i  = 0;
const int ci = -1;

i=1;
ci=22;
```

~~ci=22;~~ assignment of read-only variable `ci'

```
int* p1;

p1 = &i;
*p1=11;
p1 = &ci;
*p1 =22;
```

depending on compiler and compiler options

`p1 = &ci; *p1 =22;` → only warning: discards const

```
cout<<"ci="<<ci<<", *p1 = "<<*p1<<endl;
```

⇒ ci=-1, *p1 = 22

const and pointer (2)

```
int i  = 0; const int ci = -1;
```

```
const int* p2;
p2 = &i;
*p2=111;       // struck through
p2 = &ci;
*p2 =222;      // struck through
```

~~`*p2=111;`~~ assignment of read-only location

~~`*p2 =222;`~~ assignment of read-only location

```
int* const p3;     // struck through
int* const p4 = &i;
*p4=1111;
p4 = &ci;          // struck through
```

~~`int* const p3;`~~ uninitialized const `int * const p3'

~~`p4 = &ci;`~~ assignment of read-only variable `p4'

const and pointer (3)

```
int i  = 0; const int ci = -1;
```

```
const int* const p5;          // struck through: uninitialized const
const int* const p6 = &i;
*p6=111111;                   // struck through: assignment of read-only location
p6=&ci;                       // struck through: assignment of read-only variable `p6'
const int* const p7 = &ci;
*p7=2222222;                  // struck through: assignment of read-only location
p7=&i;                        // struck through: assignment of read-only variable `p7'
```

uninitialized const

assignment of read-only location

assignment of read-only variable `p6'

assignment of read-only location

assignment of read-only variable `p7'