
2022
To Pass and Return
The Story of Functions, Values and Compilers
Dawid Zalewski

So many possibilities…

```
void by_cref(const string& str){
    printf("%s", str->c_str());
}

int main(){
    string str{"Alice learns C++"};
    by_cref(str);
}
```

So many possibilities…

By lvalue ref

```
void by_cref(const string& str);
void by_ref(string& str);
```

1

By rvalue ref

```
void by_crref(const string&& str);
void by_rref(string&& str);
```

2

By pointer

```
void by_cptr(const string* str);
void by_ptr(string* str);
```

3

By value

```
void by_val(const string str);
void by_val(string str);
```

4

It depends...

Which is the slowest? … it depends

- For std::string (and any non-trivial string)

```
by_doesnt_matter(string):
        sub     rsp, 8
        mov     rsi, QWORD PTR [rdi]
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        add     rsp, 8
        ret
```

pointer to string's cha
data

format string "%s"

The caller will create a copy of str before calling by_val!

Which is the slowest? … it depends

```
struct string{
    char str_[32];
    const char* c_str() const { return str_; }
};
```

(The simplest, trivial string with value semantics)

Passing by value for non-believers

Some popular compilers*:

```
by_doesnt_matter(string):
        sub     rsp, 8
        lea     rsi, [rsp+16]
        mov     edi, offset .L.str.2
        xor     eax, eax
        call    printf
        add     rsp, 8
        ret
```

pointer to char array of string

format string "%s"

*your assembly may vary—but the gist is the same

Passing by value for non-believers

Some compilers optimize-away a copy of a trivial object passed to a function

Only one string is created

```
main: # @main
  sub rsp, 40
  movups xmm0, xmmword ptr [rip + .L__const.main.str+16]
  movups xmmword ptr [rsp + 16], xmm0
  movups xmm0, xmmword ptr [rip + .L__const.main.str]
  movups xmmword ptr [rsp], xmm0
  call by_val(string)
  xor eax, eax
  add rsp, 40
  ret
```

Takeaway 1&2

Trivial objects are easier to optimize

Compilers optimize aggressively beyond what the standard dictates

Before we begin

Compilers:

- gcc 12.1 (x86-64)
- clang 14.0.0 (x86-64)
- icc 2021.5.0 (x86-64)
- msvc v19.32 (x86-64)

https://godbolt.org/

Flags:

- gcc, clang, icc: **`-std=c++20 -O3 -Wall -Wextra –pedantic`**
- msvc: **`/std:c++20 /O2 /W4 /WX /GS- /permissive-`**

To pass and return

```
void function(T val);
T&& function();
void function(T* ptr);
void function(const T& ref);
T function();
void function(const T&& ref);
```

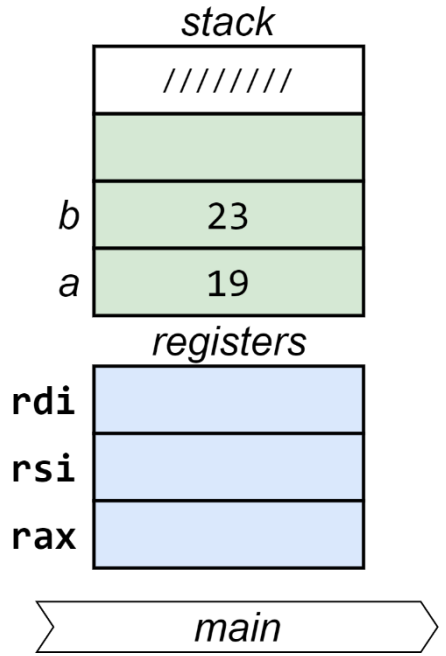
```
void function(T& ref);
void function(const T* ptr);
T& function();
void function(T&& ref);
T* function();
```

Memory model & ABI 101

```
long add(long a, long b){
  auto sum{ a + b };
  return sum;
}

int main(){
  long a{19};
  long b{23};

  long sum{ add( a, b )};

  printf("%ld", sum);
}
```

Memory model & ABI 101

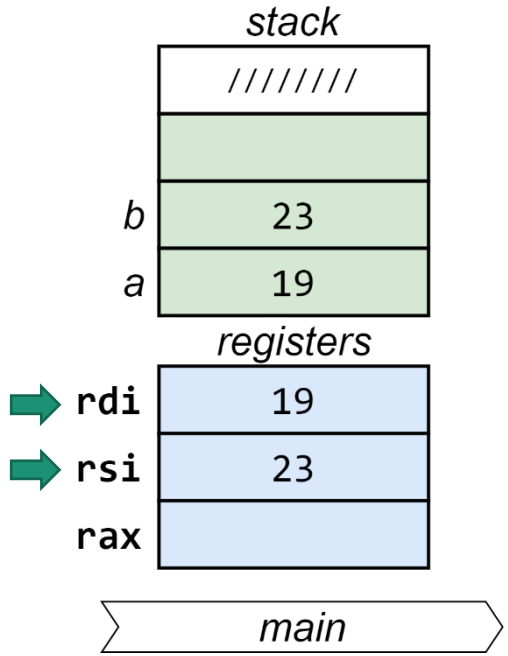
```
long add(long a, long b){
  auto sum{ a + b };
  return sum;
}

int main(){
  long a{19};
  long b{23};

  long sum{ add( a, b )};

  printf("%ld", sum);
}
```



- initializes variables on the stack

Memory model & ABI 101

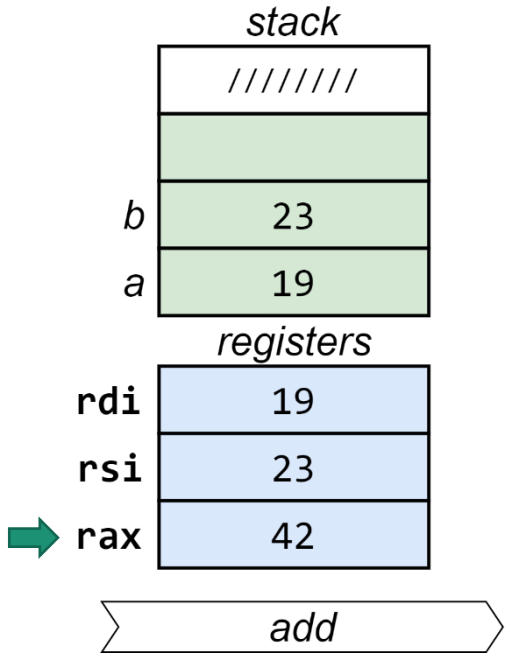
```
long add(long a, long b){
  auto sum{ a + b };
  return sum;
}

int main(){
  long a{19};
  long b{23};

  long sum{ add( a, b )};

  printf("%ld", sum);
}
```



- copies values of the variables into registers

Memory model & ABI 101

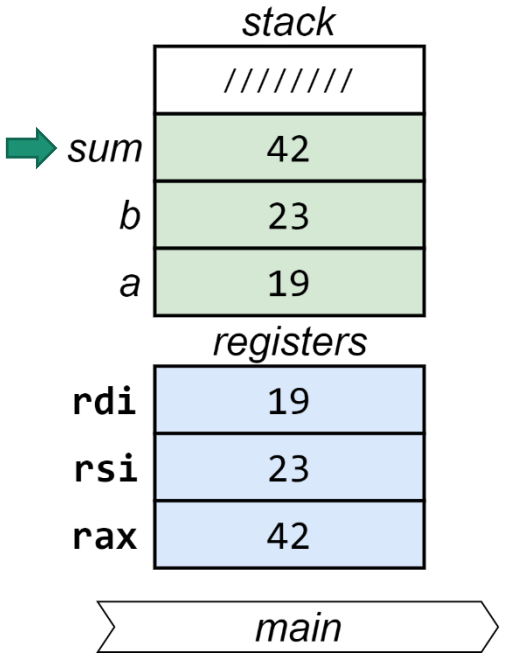
```
long add(long a, long b){
  auto sum{ a + b };
  return sum;
}

int main(){
  long a{19};
  long b{23};

  long sum{ add( a, b )};

  printf("%ld", sum);
}
```



- adds the numbers and puts the result in **rax**

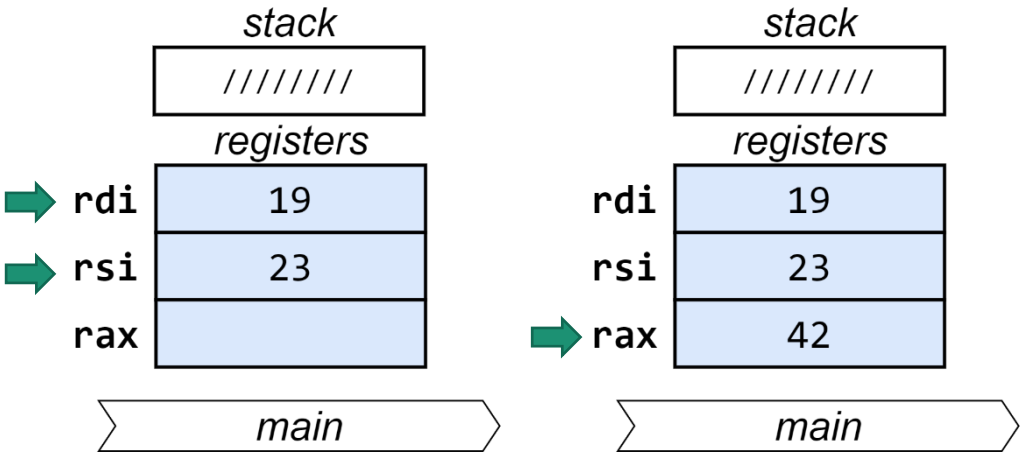
Memory model & ABI 101

```
long add(long a, long b){
  auto sum{ a + b };
  return sum;
}

int main(){
  long a{19};
  long b{23};

  long sum{ add( a, b )};

  printf("%ld", sum);
}
```



- initializes **sum** on the stack with the value in **rax**

Memory model & ABI 101 (with any optimization)



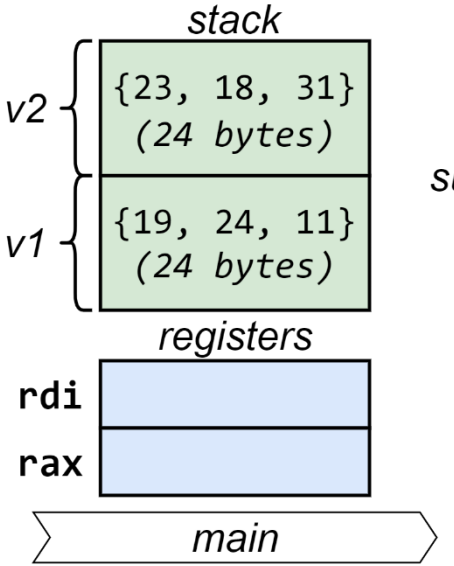
- initializes variables in the registers
- adds the numbers (**add** is inlined)

Memory model & ABI 101

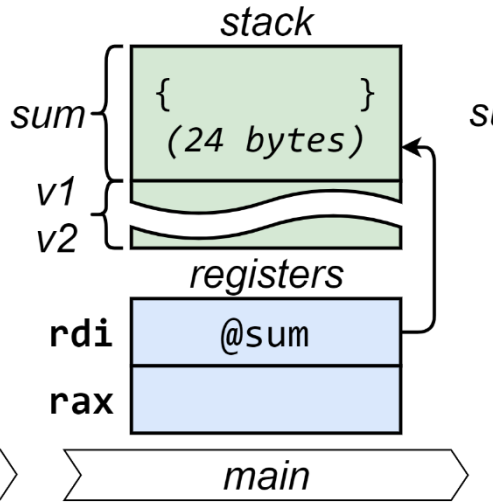
```
struct vec3d{
    long x, y, z;
};

vec3d add( vec3d a,  vec3d b){
    return { a.x+b.x, a.y+b.y, a.z+b.z};
}

int main(){
  vec3d v1{19, 24, 11}, v2{23, 18, 31};

  vec3d sum{ add( v1, v2 )};

  printf("%ld, %ld, %ld", sum.x, sum.y, sum.z);
}
```

Returns an “oversized” value

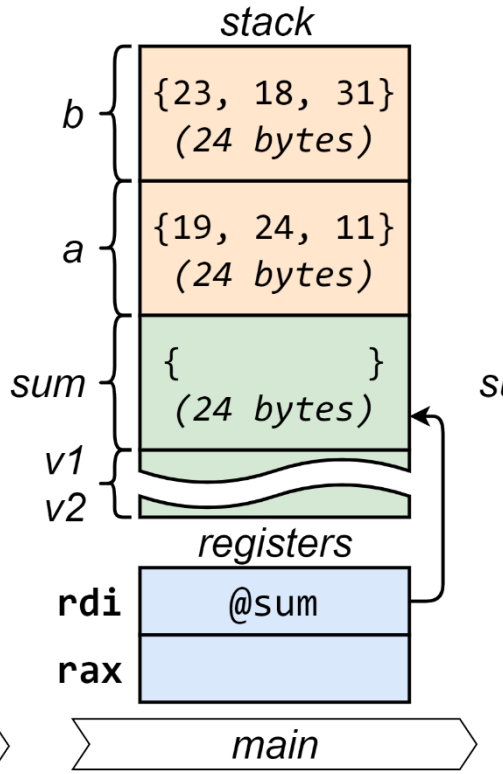
Memory model & ABI 101



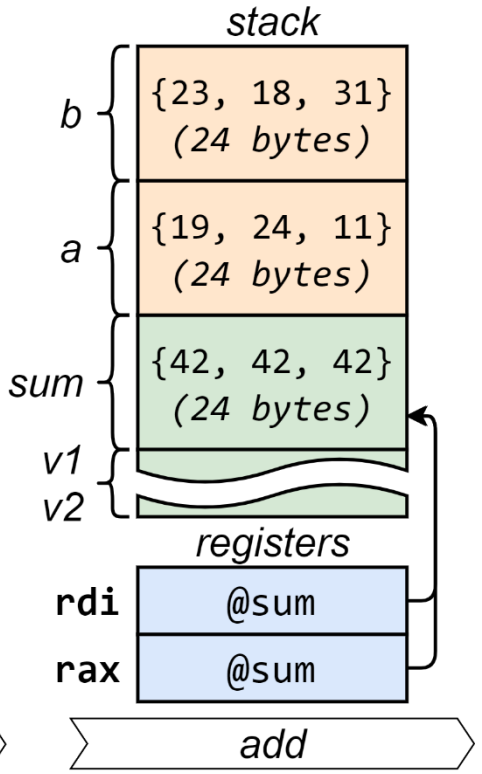
- initializes variables on the stack



- creates space for the oversized return object on the stack
- puts pointer to it in **rdi**



- creates arguments on the stack for oversized objects by copying **v1** & **v2**



- adds the vectors
- returns pointer to the result in **rax**

Takeaway 3

EVERYTHING IS PASSED AND RETURNED BY VALUE

Takeaway 3

EVERYTHING IS PASSED AND RETURNED BY VALUE

- A full binary representation of an object
- A memory address of a binary representation of an object

Memory model & ABI 101

What	System V AMD64	Microsoft x64
1st argument	`rdi`	`rcx`
2nd argument	`rsi`	`rdx`
3rd argument	`rdx`	`r8`
4th argument	`rcx`	`r9`
5th argument	`r8`	`stack`
6th argument	`r9`	`stack`
Return value	`rax`	`rax`

Also used for passing an address to a big return object

- Registers: objects with sizes not greater than 64 bits (integers, pointers).
- Stack: objects that do not fit in registers.

Returning, by value

Returning, by value

Return by-value (usually) means:

```
SomeType function(){
  /* ~~~ */
  return { /* ~~~ */ };
}

/* ~~~ */

auto obj { function() };
```

Then, a copy of it is made here

A SomeType (temporary) object is created here

And finally one more copy when initializing obj from the return value

Objects as return values

It all depends on the compiler one uses, but I know that at least the AT&T cfront and GNU C++ are smarter than this. In these compilers, the caller passes the address of the place where the new temporary should be initialized. Depending on the way it is initialized, there may be no overhead visible from the call to `operator +` at all:

```
M operator + (M x, M y)
{
  return M (x.value () + y.value ());
}
```

Objects as return values, Michael Tiemann in *C++ Gems* (1998)

Objects as return values

It is frequently possible to write functions that return objects in such a way that compilers can eliminate the cost of the temporaries. The trick is to return constructor arguments instead of objects (...)

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

Item 20: Facilitate the return value optimization,
Scott Meyers in *More Effective C++* (1995)

Objects as return values

It is frequently possible to write functions that return objects in such a way that compilers can eliminate the cost of the temporaries. The trick is to return constructor arguments instead of objects (...)

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
  return {lhs.numerator() * rhs.numerator(),
          lhs.denominator() * rhs.denominator()};
}
```

Item 20: Facilitate the return value optimization,
Scott Meyers in *More Effective C++* (1995)

Returning non-trivial objects

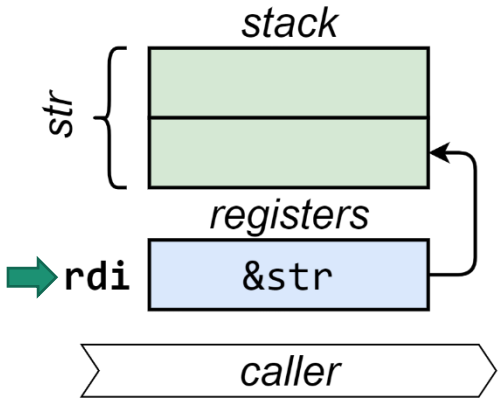
A *non-trivial* test type:

```
struct proper_string {
  proper_string();
  proper_string(const char*)


  /* +rule of five */

  const char* c_str() const;

  std::size_t len_;
  char* str_;
};
```

- *copy constructor*
- *copy assignment operator*
- *move constructor*
- *move assignment operator*
- *destructor*

Returning, by value (non-trivial type)

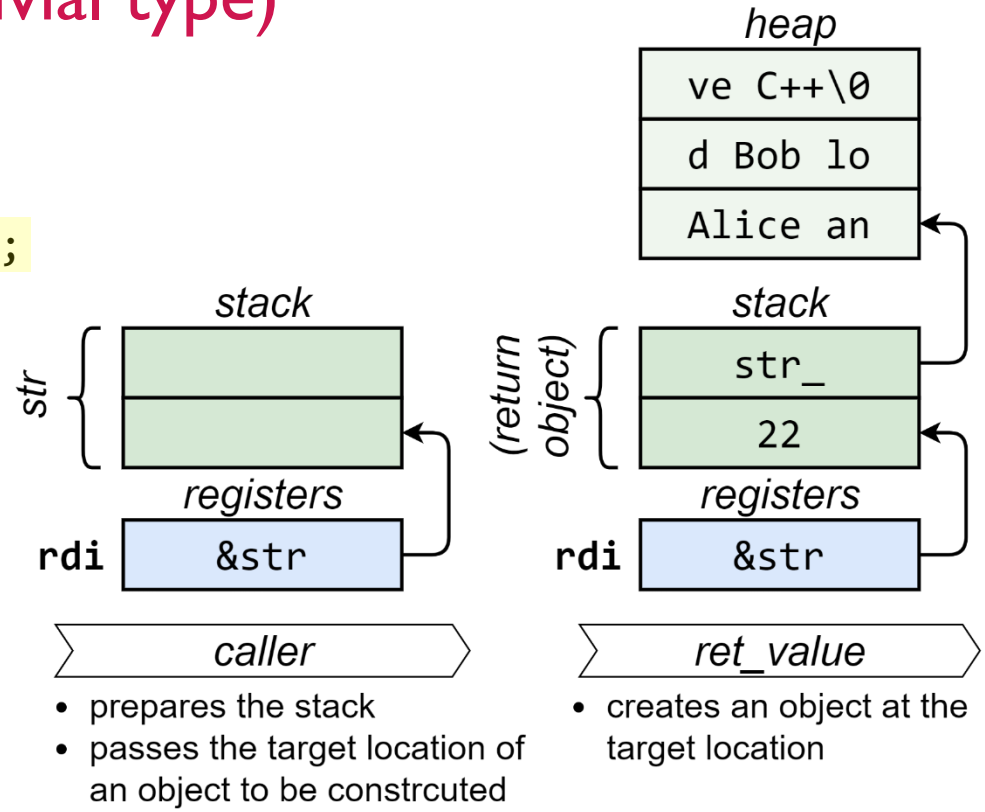
```
proper_string ret_value(){
  return {"Alice and Bob love C++"};
}

auto str{ ret_value() };
printf("%s", str.c_str());
```



- prepares the stack
- passes the target location of an object to be constrcuted

Returning, by value (non-trivial type)

```
proper_string ret_value(){
  return {"Alice and Bob love C++"};
}

auto str{ ret_value() };
printf("%s", str.c_str());
```



- prepares the stack
- passes the target location of an object to be constrcuted

- creates an object at the target location

Returning, by value – copy elision

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial	✓	✓	✓	✓

✓ – full copy/ move elision.

Returning, by value (trivial type)

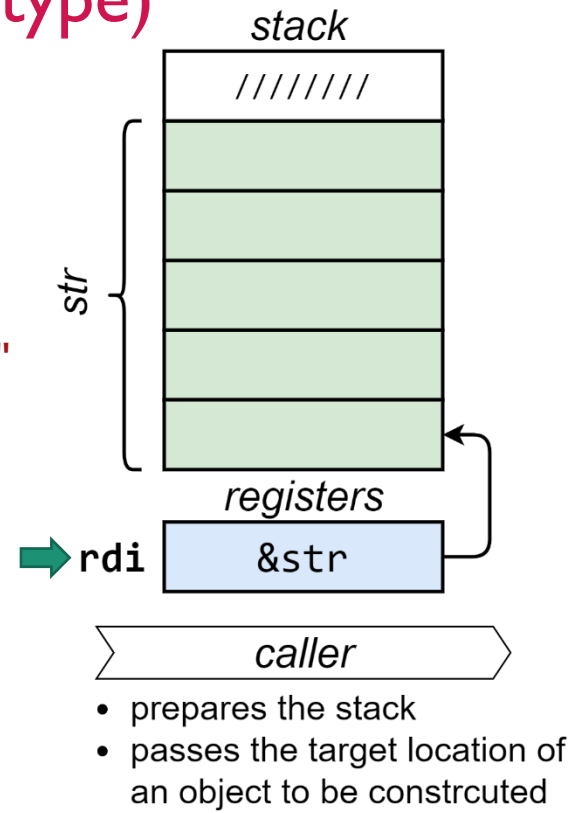
A *trivial* test type:

```
struct trivial_string{
    std::size_t len_;
    char str_[SZ_MAX];
    const char* c_str() const { return &str_[0]; }
};
```

- `std::is_aggregate_v<trivial_string>` ✓
- `std::is_trivial_v<trivial_string>` ✓
- `is_oversized<trivial_string>` ✓

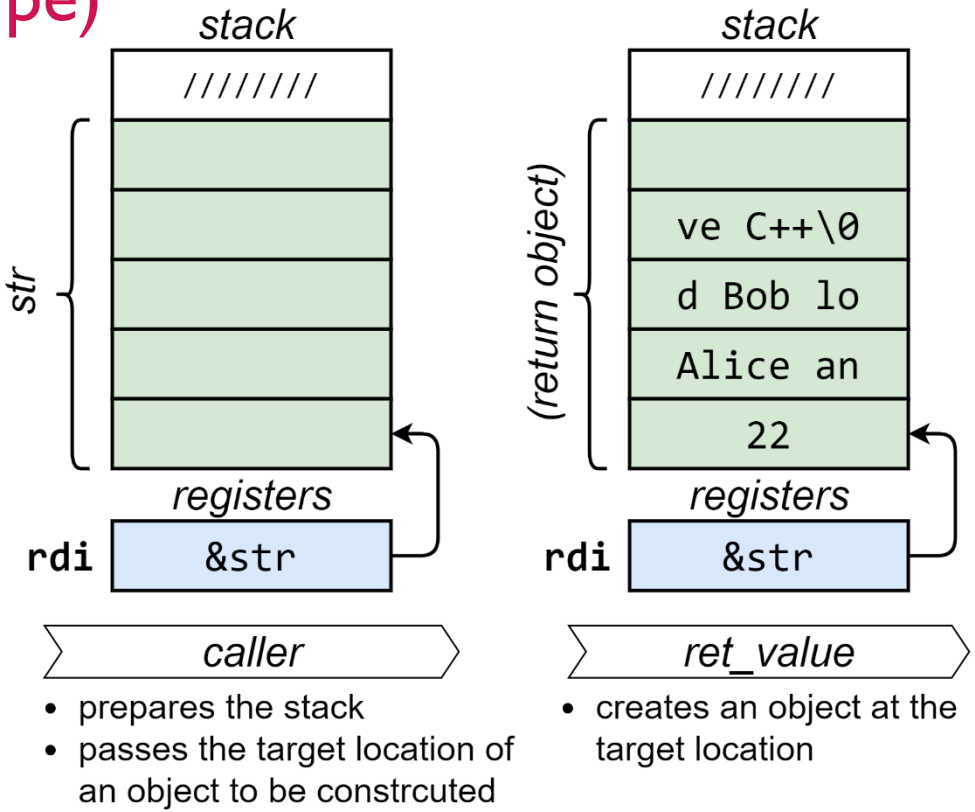
Returning, by value (trivial type)

```
trivial_string ret_value(){
  return {
     .len_=22,
     .str_="Alice and Bob love C++"
  };
}

auto str{ ret_value() };
printf("%s", str.c_str());
```



Returning, by value (trivial type)

```
trivial_string ret_value(){
  return {
     .len_=22,
     .str_="Alice and Bob love C++"
  };
}

auto str{ ret_value() };
printf("%s", str.c_str());
```



- prepares the stack
- passes the target location of an object to be constrcuted

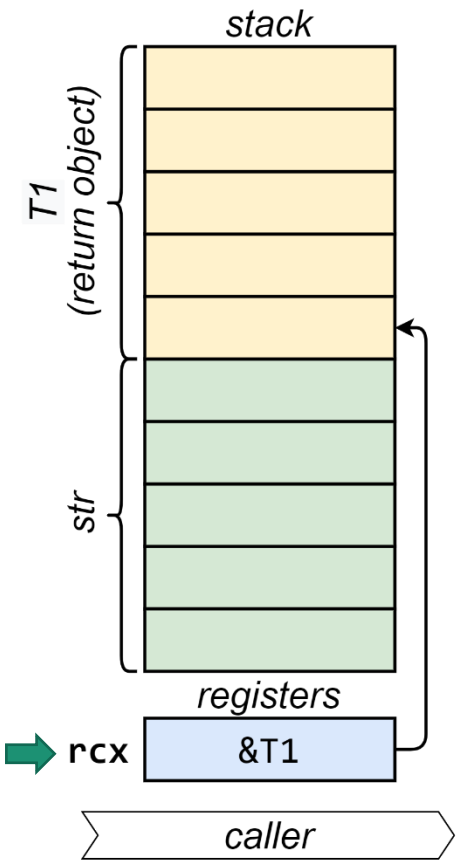
- creates an object at the target location

Returning, by value (trivial type)

All compilers agree:

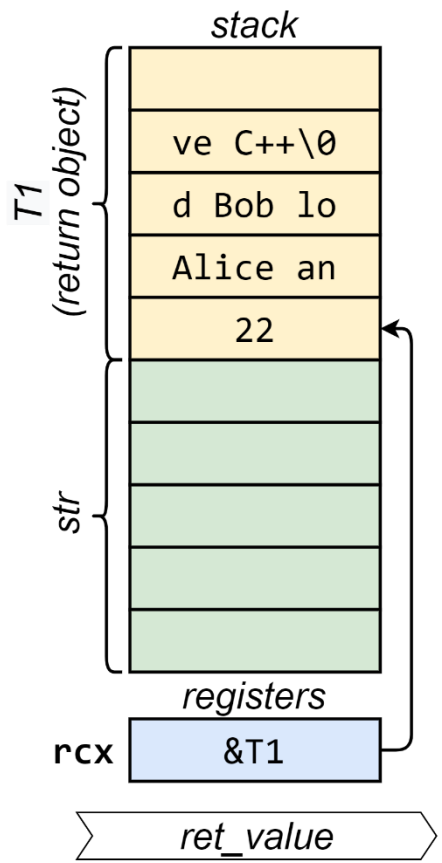
- object created by the callee directly on the stack at the *target location*
- copy elision (known as *return value optimization* – RVO)

(All compilers besides …
… MSVC)



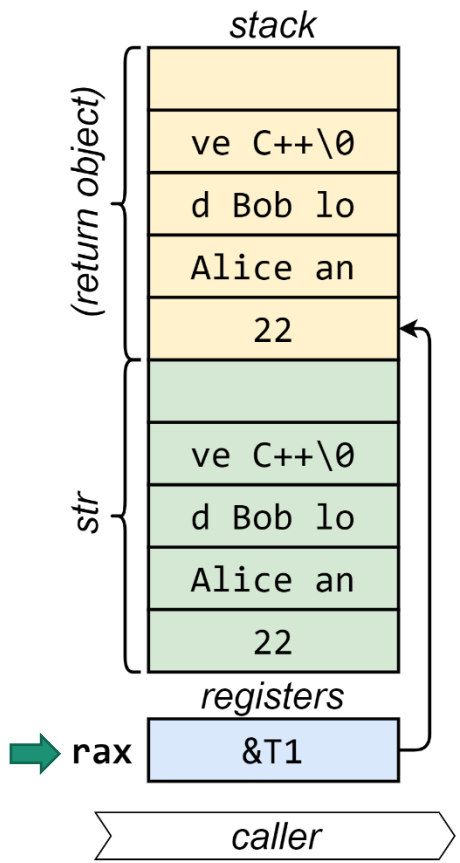
caller

- prepares the stack
- passes the target location of a temporary object (**T1**) to be constrcuted



ret_value

- creates an object at the target location



caller

- copy-constrcuts **str** from the temporary **T1**

Returning, by value – copy elision

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial	✓	✓	✓	✓
	Trivial	✓	✓	✓	😱

✓ – full copy/ move elision.

😱 – one copy-construction after callee returns.

Takeaway 4

Don't trust your compiler (blindly)

Returning – copy elision

Then, a copy/ move of it is elided here

```
SomeType function(){
  /* ~~~ */
  return { init_args };
}

/* ~~~ */

auto obj { function() };
```

A SomeType (temporary) object is created here

And once more here when initializing obj

Only possible when copy ctor/ move ctor exist.

Returning – delayed temporary materialization


```
SomeType function(){
  /* ~~~ */
  return { init_args };
}

/* ~~~ */

auto obj { function() };
```

...passed here but since that's not the final stop...

Nothing is created here, init_args are magically...

...they are passed one step further to initialize obj

Enabled by delayed temporary materialization in C++17.

Returning, by value – copy elision

```
T function(){
  return T{};
}
```

When	What	C++11	C++17
	Copy/ Move elision	Optional	Mandatory
	`T(const T&), T(T&&)`	Must be present	Optional
	Side effects of `T(const T&), T(T&&)`	Ignored	Ignored

Returning, by value – copy elision

When	What	C++11	C++17
`T function(){ return T{}; }`	**Copy/ Move elision**	**Optional**	**Mandatory**
	`T(const T&), T(T&&)`	**Must be present**	**Optional**
	Side effects of `T(const T&), T(T&&)`	**Ignored**	**Ignored**
`T function(){ T obj{}; return obj; }`	**Copy/ Move elision**	**Optional**	**Optional**
	`T(const T&), T(T&&)`	**Must be present**	**Optional**
	Side effects of `T(const T&), T(T&&)`	**Ignored**	**Ignored**

Returning, by value II (non-trivial)

```
proper_string ret_value(){
  proper_string result{"Alice and Bob love C++"};
  if (random_condition()){
    result = "Alice and Bob like C!";
    return result;
  }
  return result;
}

auto str{ ret_value() };
printf("%s", str.c_str());
```

A named object result is created

Two return statements

Objects as return values

A **really smart compiler** could notice that *result* was only feeding the return value, and substitute it for *result* throughout.

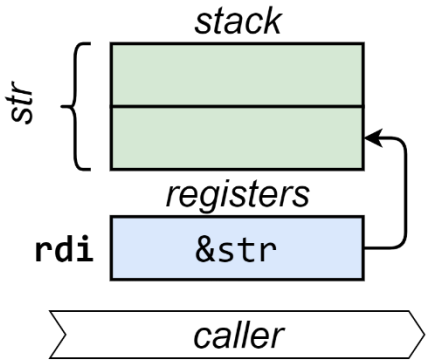
Another solution might be to extend the language:

```
trivial_string ret_value ()
  return result;
{
    /* ~~~ */
}
```

Objects as return values, Michael Tiemann in *C++ Gems* (1998)

Returning, by value II (non-trivial)

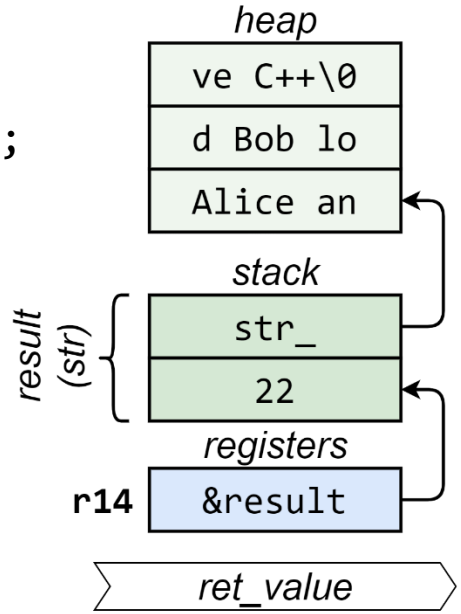
```
proper_string ret_value(){
  proper_string result{"Alice and Bob love C++"};
  if (random_condition()){
    result = "Alice and Bob like C!";
    return result;
  }
  return result;
}

auto str{ ret_value() };
printf("%s", str.c_str());
```



- prepares the stack
- passes the target location of an object to be constrcuted

Returning, by value II (non-trivial)

```
proper_string ret_value(){
  proper_string result{"Alice and Bob love C++"};
  if (random_condition()){
    result = "Alice and Bob like C!";
    return result;
  }
  return result;
}

auto str{ ret_value() };
printf("%s", str.c_str());
```



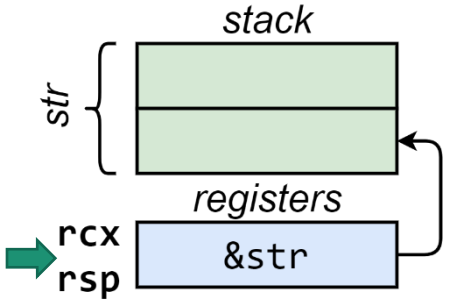
Returning, by value II (non-trivial)

Most compilers agree on full copy elision…

…msvc does something else.

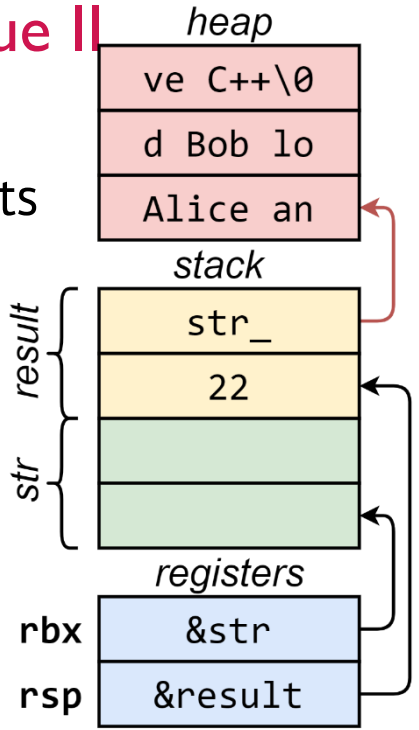
Returning, by value II (non-trivial)

msvc move-constructs from return object



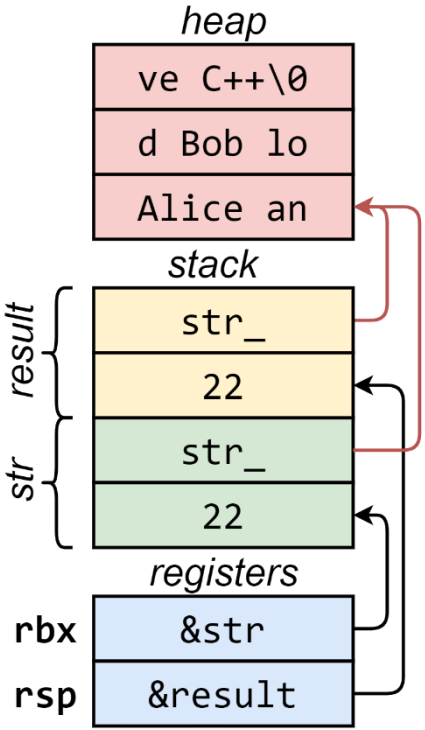
caller

- prepares the stack
- passes the target location of an object to be constrcuted



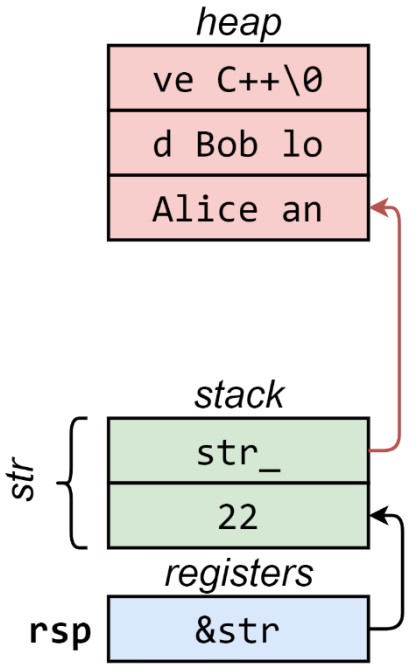
ret_value

- adjusts the stack
- creates an object in its own stack space



ret_value

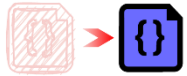
- moves the object from its own stack space to the target location



caller

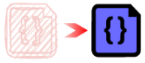
- copy elided

Returning, by value – copy elision

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial	✓	✓	✓	✓
	Trivial	✓	✓	✓	😱
`T function(){ T obj{}; return obj; }`	Non-trivial	✓	✓	✓	

✓ – full copy/ move elision.

😱 – copy-construction after callee returns.

 – move-construction from named lvalue by callee.

Returning, by value II (trivial)

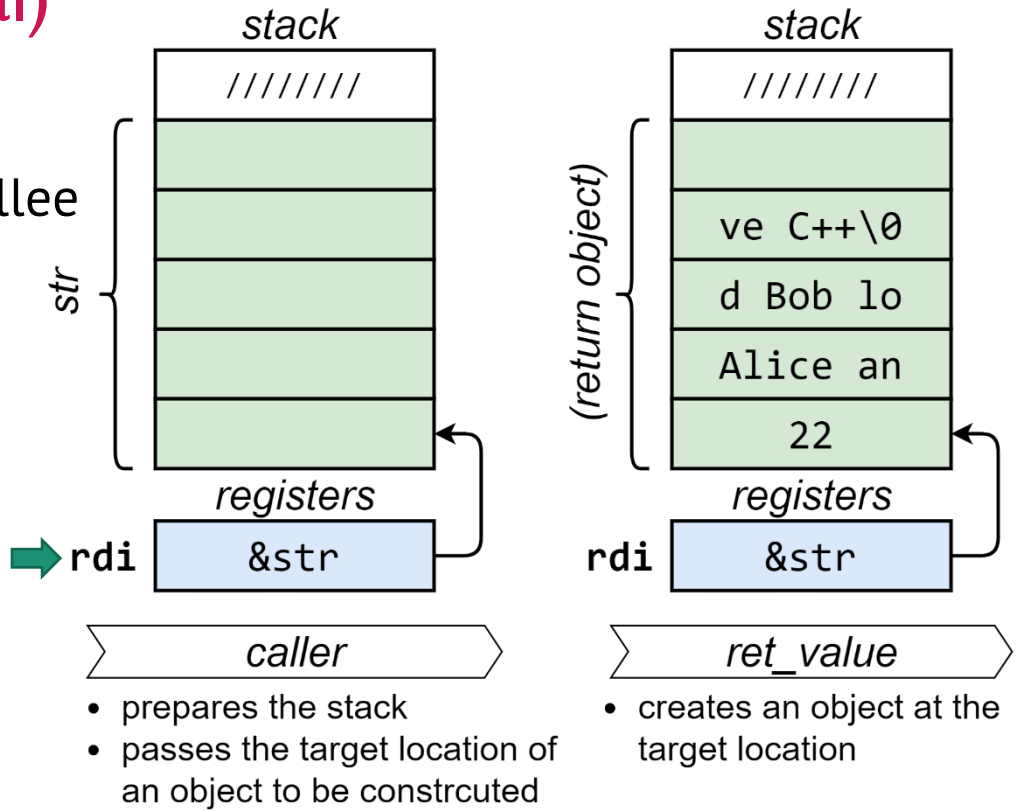
```
trivial_string ret_value(){
  trivial_string result{22, "Alice and Bob love C++"};
  if (random_condition()){
    result = {21, "Alice and Bob like C!"};
    return result;
  }
  return result;
}

auto str{ ret_value() };
printf("%s", str.c_str());
```

```
struct trivial_string{
  std::size_t len_;
  char str_[SZ_MAX];
  const char* c_str() const { return &str_[0]; }
};
```

Returning, by value II (trivial)

Some compilers* agree:

- Object (`result`) created by the callee directly on the stack at the target location
- Full copy elision

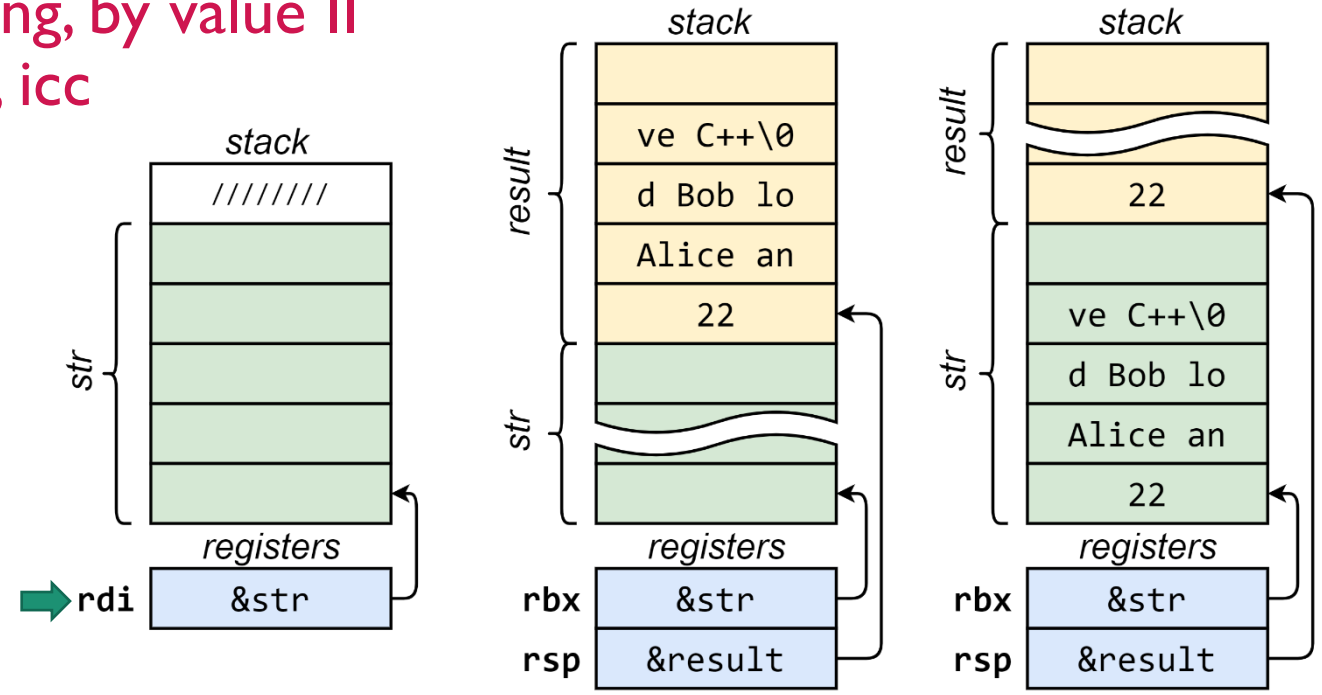


- prepares the stack
- passes the target location of an object to be constrcuted

- creates an object at the target location

**some compilers: gcc & clang*

Returning, by value II (trivial), icc

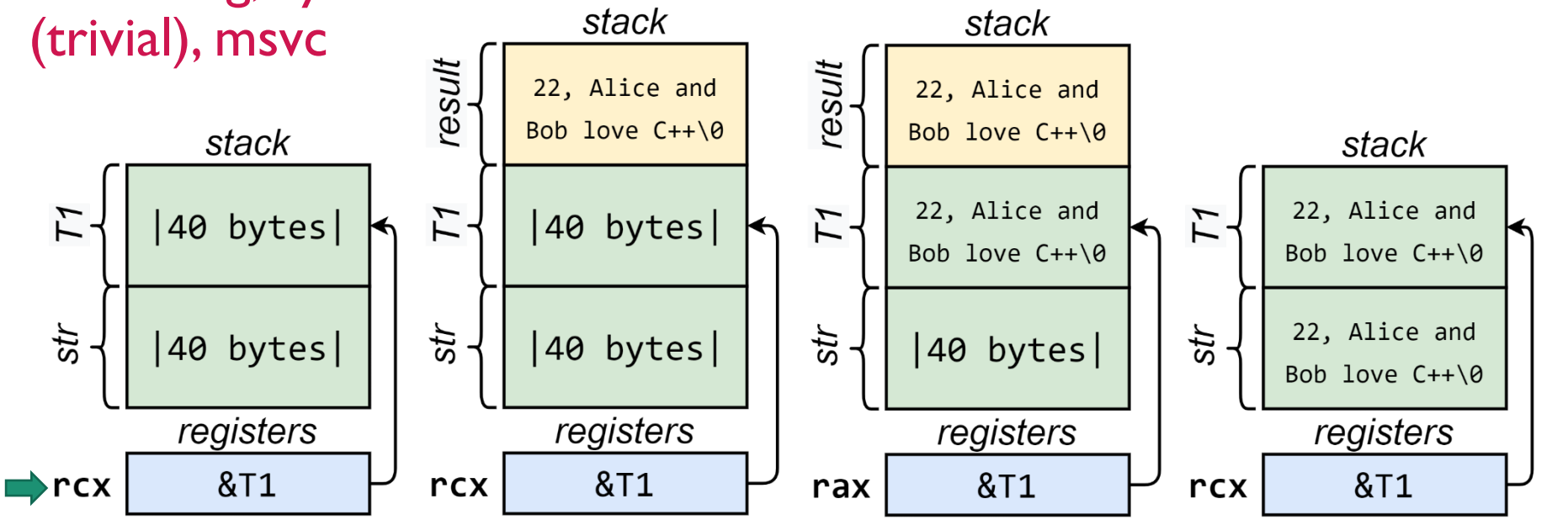


- prepares the stack
- passes the target location of an object to be constrcuted

- adjusts the stack
- creates a new object in its own stack space

- copies the object to its target destination

Returning, by value II (trivial), msvc



caller

- prepares the stack
- passes the target location of a return object (**T1**)

ret_value

- adjusts the stack
- constructs a named lvalue (**result**) in its stack space

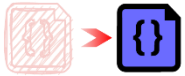
ret_value

- copy-constructs the return object (**T1**) from **result**

caller

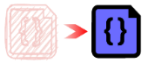
- copy-constructs **str** from the return object (**T1**)

Returning, by value – copy elision

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial	✓	✓	✓	✓
	Trivial	✓	✓	✓	
`T function(){ T obj{}; return obj; }`	Non-trivial	✓	✓	✓	
	Trivial	✓	✓		2x 

✓ – full copy/ move elision.

 – copy-construction after callee returns.

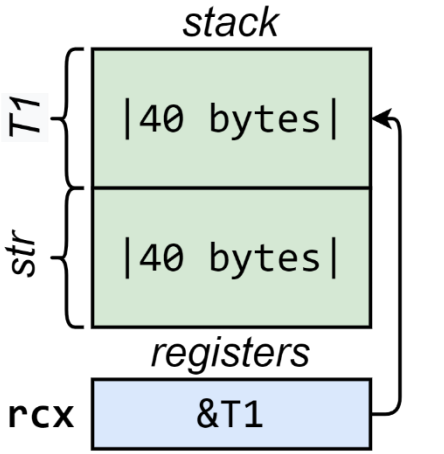
 – move-construction from named lvalue by callee.

 – copy construction.

Returning, by value II (trivial)

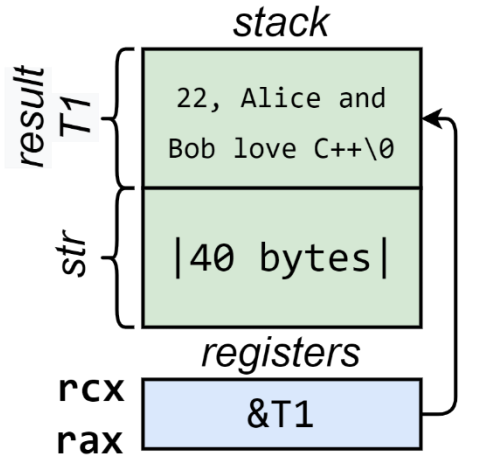
```
trivial_string ret_value(){
  trivial_string result{22, "Alice and Bob love C++"};
  if (random_condition()){
    result = {21, "Alice and Bob like C!"};
    return result;
  }
  return result;
}

auto str{ ret_value() };
printf("%s", str.c_str());
```

Returning, by value II (trivial), msvc–single return*



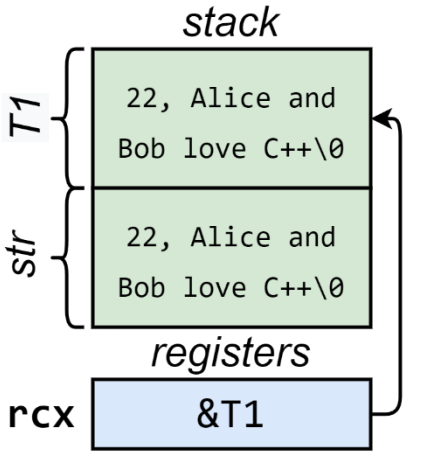
caller

- prepares the stack
- passes the target location of a return object (**T1**)



ret_value

- copy-constructs the return object (**T1**) directly

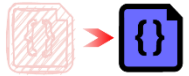


caller

- copy-constructs **str** from the return object (**T1**)

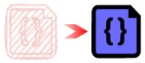
**Only when the if statement tests negative.*

Returning, by value – copy elision

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial	✓	✓	✓	✓
	Trivial	✓	✓	✓	😱
`T function(){ T obj{}; return obj; }`	Non-trivial	✓	✓	✓	
	Trivial	✓	✓		2x 

✓ – full copy/ move elision.

😱 – copy-construction after callee returns.

 – move-construction from named lvalue by callee.

 – copy construction.

*Only **1.5x** with single return*

Takeaway 5

Single return is (still) your friend

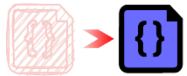
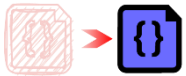
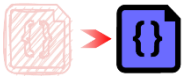
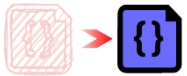
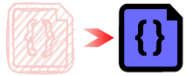
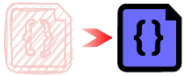
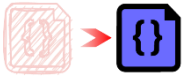
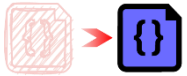
Assigning from function call, by value

Returning, by value

```
some_string ret_value(){
  some_string result{"Alice and Bob love C++"};
  /* ~~~ */
  return result;
}

some_string str{"Hello World of C++"};
str = ret_value();

printf("%s", str.c_str());
```

Return value used in assignment

Returning, by value – copy elision (assignment)

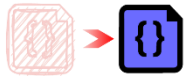
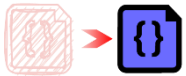
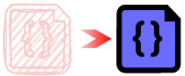
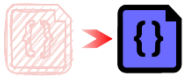
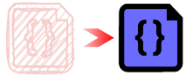
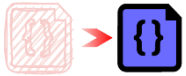
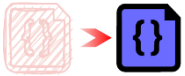
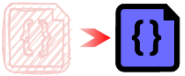
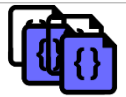
When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial				
	Trivial				
`T function(){ T obj{}; return obj; }`	Non-trivial				2x 
	Trivial			2x 	2x 

✓ – full copy/ move elision

 – copy assignment (construction)

 – move assignment (construction)

Returning, by value – copy elision (assignment)

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial				
	Trivial	✓	✓	✓	
`T function(){ T obj{}; return obj; }`	Non-trivial				2x 
	Trivial				2x 

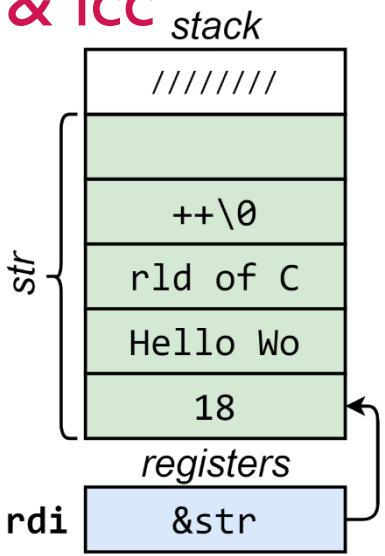
✓ – full copy/ move elision

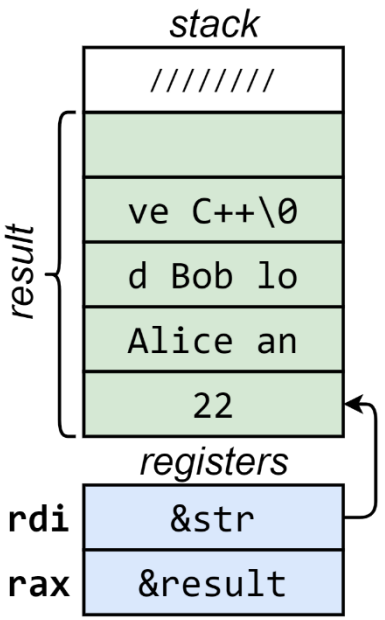
 – copy assignment (construction)

 – move assignment (construction)

Returning, by value, gcc, clang & icc

```
trivial_string ret_value(){
  return {
          22, "Alice and Bob love C++"
         };
}

trivial_string str{
           18, "Hello World of C++"};

str = ret_value();

printf("%s", str.c_str());
```





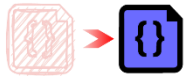
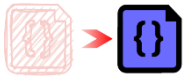
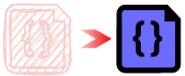
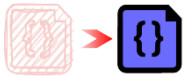
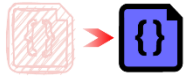
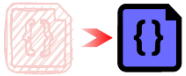
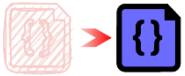
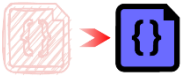
caller

- initializes **`str`** object
- passes the target location (**`&str`**) to the callee

ret_value

- creates the result/ return object at the target location (overwriting previously held value)

Returning, by value – copy elision (assignment)

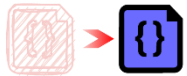
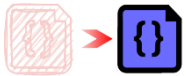
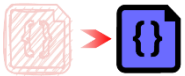
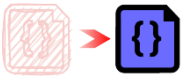
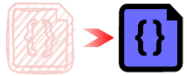
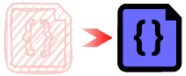
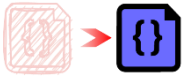
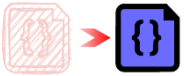
When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial				
	Trivial	✓	✓	✓	
`T function(){ T obj{}; return obj; }`	Non-trivial				2x 
	Trivial				2x 

✓ – full copy/ move elision

 – copy assignment (construction)

 – move assignment (construction)

Returning, by value – copy elision (assignment)

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial				
	Trivial	✓	✓	✓	
`T function(){ T obj{}; return obj; }`	Non-trivial				2x 
	Trivial		✓		2x 

✓ – full copy/ move elision

 – copy assignment (construction)

 – move assignment (construction)

Takeaway 6

Big chains of copies/ moves can be optimized by really smart compilers

Returning, by value – copy elision (assignment)

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-trivial				
	Trivial	✓	✓	✓	
`T function(){ T obj{}; return obj; }`	Non-trivial				2x
	Trivial		✓		2x 

✓ – full copy/ move elision

 – copy assignment (construction)

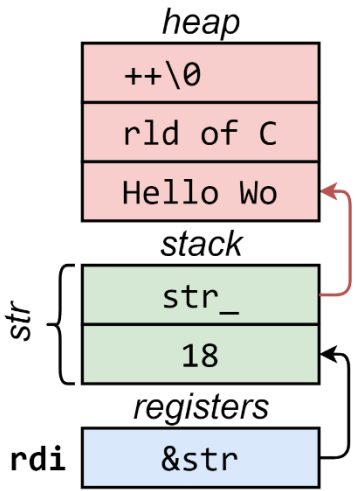
 – move assignment (construction)

Returning, by value, non-trivial, lvalue & assignment

```
proper_string ret_value(){
  proper_string result{"Alice and Bob love C++"};
  if (random_condition()){
  /* ~~~ */
}
proper_string str{"Hello World of C++"};

printf("%s", str.c_str());

str = ret_value();

printf("%s", str.c_str());
```

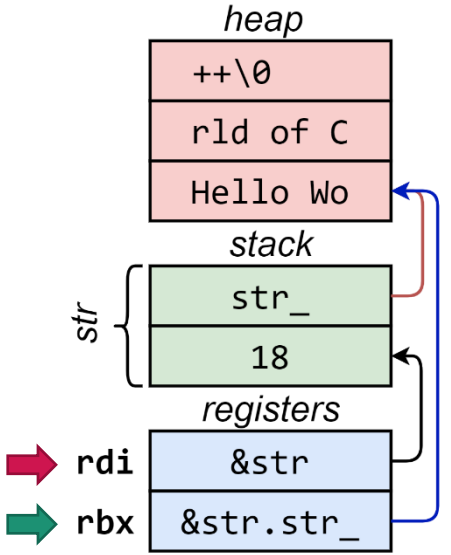
An extra challange

gcc, clang:
non-trivial, lvalue & assignment



caller

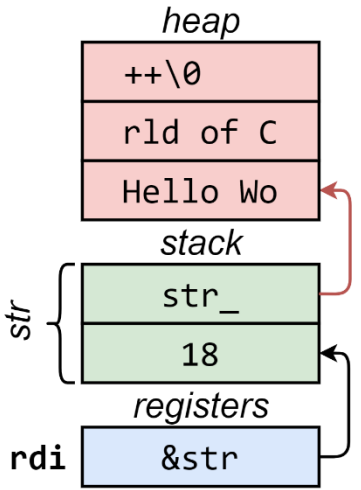
- initializes **str** object
- passes the target location (**&str**) to the callee



caller

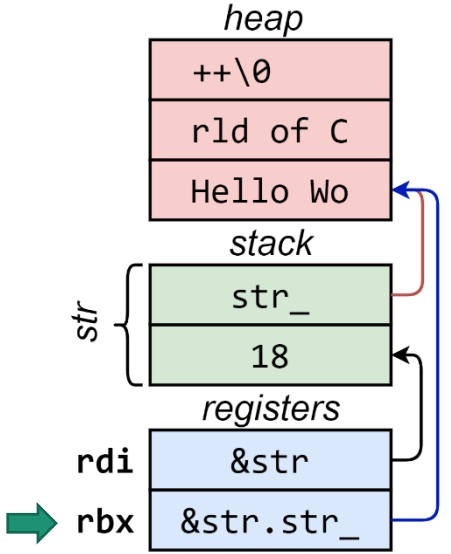
- initializes **str** object
- passes the target location (**&str**) to the callee
- saves the address of **str.str_** in **rbx**

gcc, clang:
non-trivial, lvalue & assignment



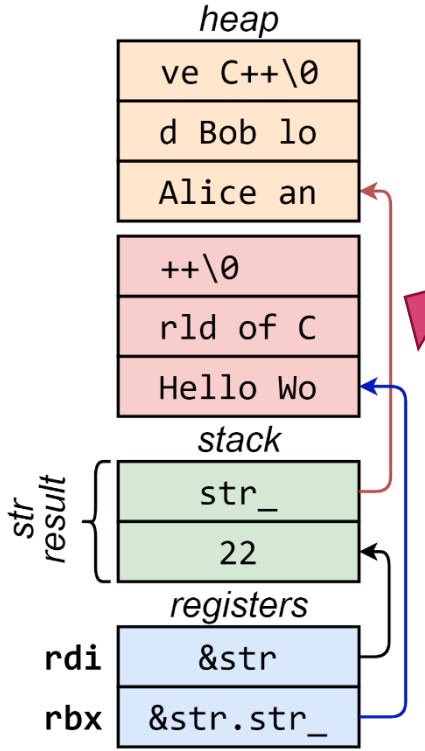
caller

- initializes **str** object
- passes the target location (**&str**) to the callee



caller

- initializes **str** object
- passes the target location (**&str**) to the callee
- saves the address of **str.str_** in **rbx**



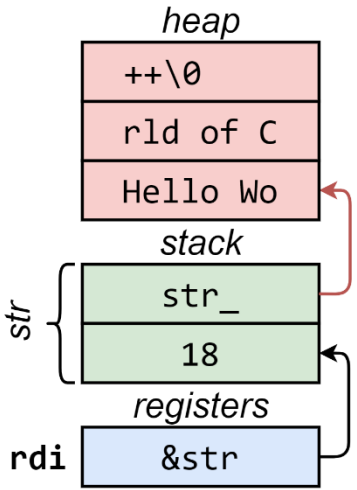
ret_val

- creates `result` at the target location (allocating new memory)

Str *now owns memory allocated by* ***ret_val***

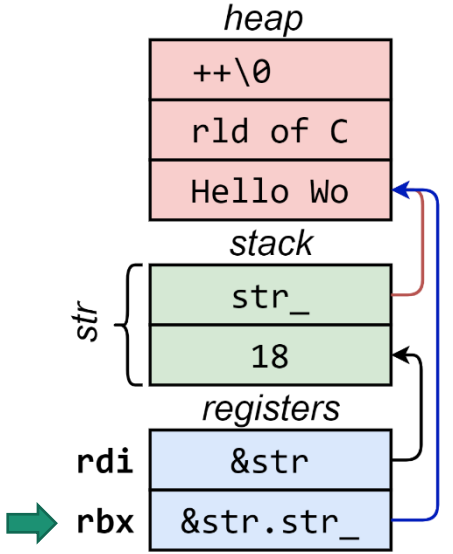
Memory that str should own is held in ***rbx***

gcc, clang: non-trivial, lvalue & assignment



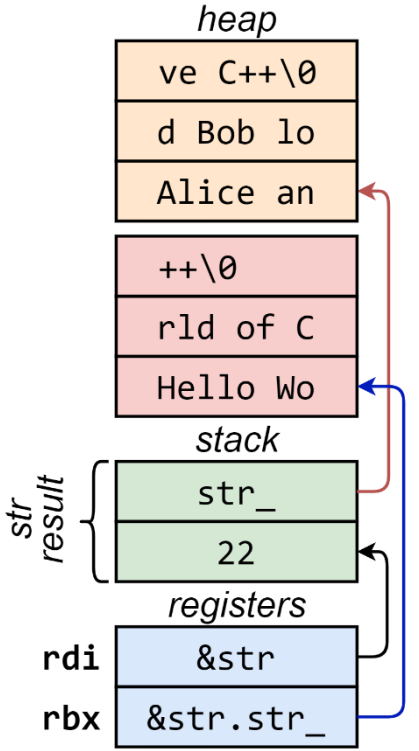
caller

- initializes **`str`** object
- passes the target location (**`&str`**) to the callee



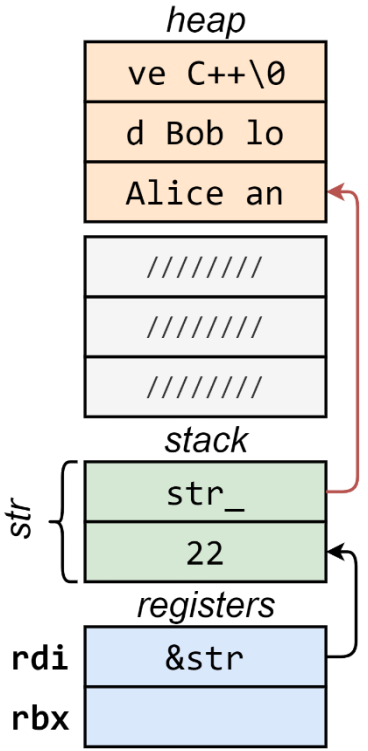
caller

- initializes **`str`** object
- passes the target location (**`&str`**) to the callee
- saves the address of **`str.str_`** in **`rbx`**



ret_val

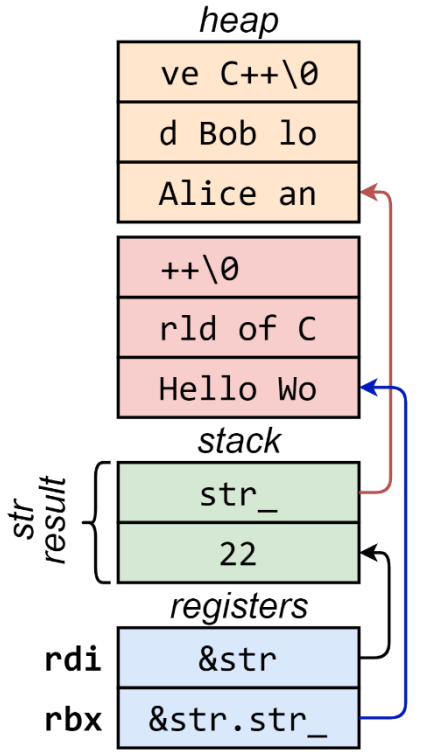
- creates `result` at the target location (allocating new memory)



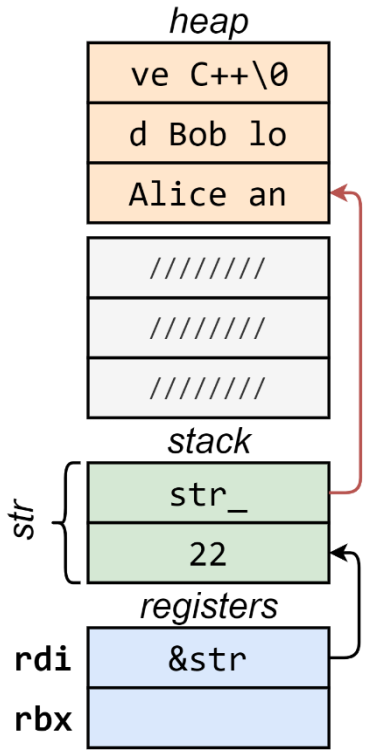
caller

- deallocates memory held in **`rbx`**

```
proper_string&
operator=(proper_string&& other){
  if (this != &other){
    ::operator delete(str_);   // caller
    len_ = other.len_;         // callee
    str_ = other.str_;
    other.len_ = 0;
    other.str_ = nullptr;
  }
  return *this;
}

void caller(){
  proper_string str{...};
  str = ret_val();
}
```



- creates `result` at the target location (allocating new memory)



- deallocates memory held in **rbx**

Takeaway 7

Compilers take advantage of the as-if rule

Neither passing, nor returning by value means always making a copy (Compilers aggressively avoid copies and moves)

Analyze the machine code, there are hidden gems there (This also helps optimizing code and avoiding nasty surprises)

Time for answers!

to pass and return –
the story of functions, values and compilers

Dawid Zalewski

github.com/zaldawid

zaldawid@gmail.com

saxion.edu