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

Dawid Zalewski

github.com/zaldawid
zaldawid@gmail.com
saxion.edu

What is it about?

AN EXPLORATION OF HOW COMPILERS SEE OBJECTS RETURNED OR PASSED TO FUNCTIONS.

HOW WE, PROGRAMMERS, COULD USE THOSE INSIGHTS.

Before we begin

Compilers:

- gcc 11.2 (x86-64)
- clang 13.0.0 (x86-64)
- icc 2021.3.0 (x86-64)
- msvc v19.29 (x64)

Flags:

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

To pass and return

Let's start with a bold statement…

EVERYTHING IS PASSED AND RETURNED BY VALUE

To pass and return

Let's start with a bold statement…

EVERYTHING IS PASSED AND RETURNED BY VALUE

A value is either:

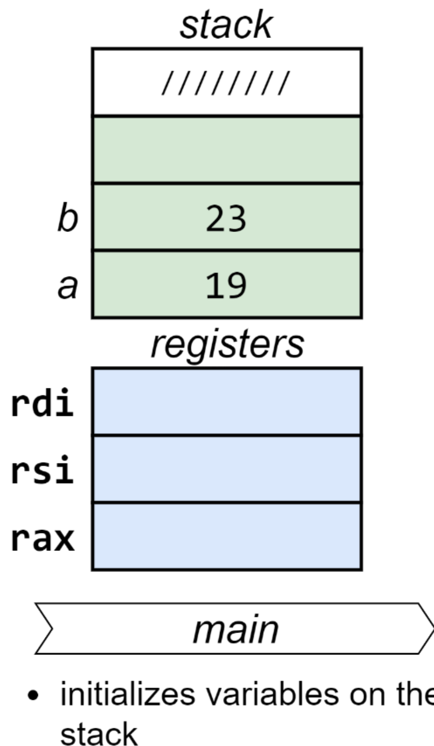
- A full binary representation of an object
- A memory address of a binary representation of an object

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

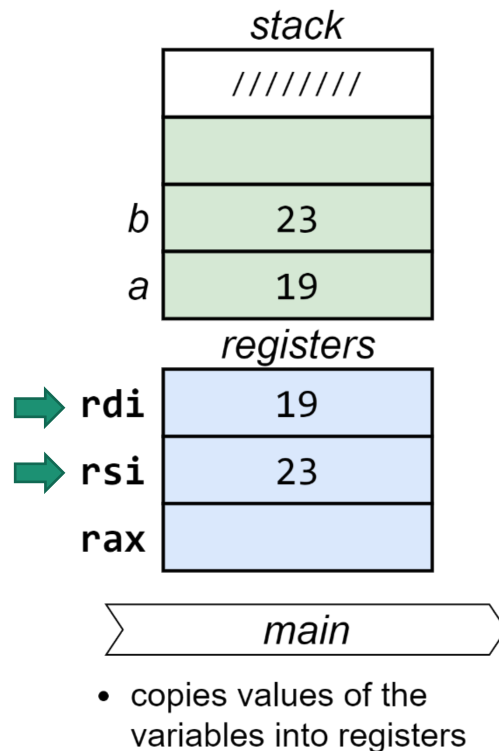
```
long add(long a, long b){
  auto sum{ a + b };
  return sum;
}

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

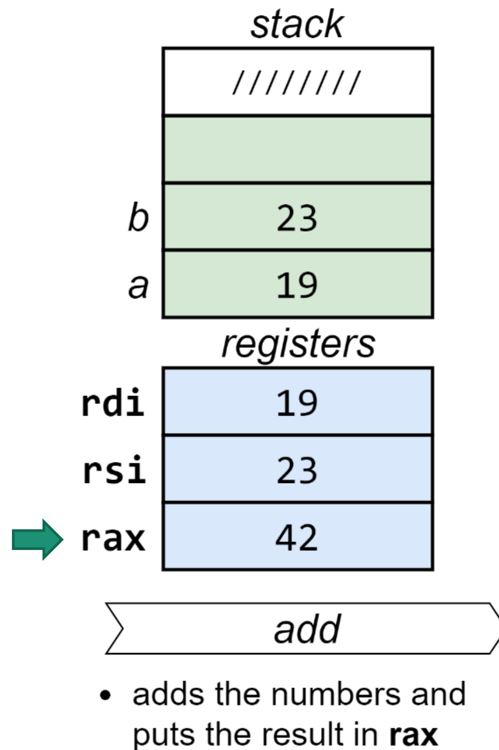
```
long add(long a, long b){
   auto sum{ a + b };
   return sum;
}

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

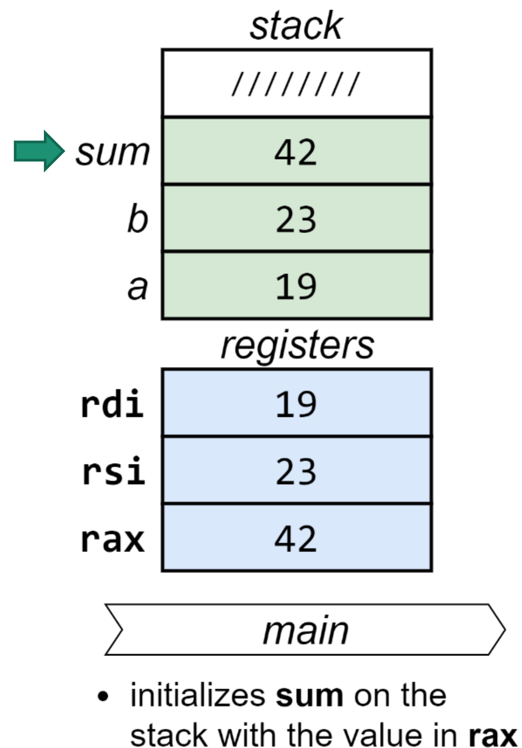
```
long add(long a, long b){
  auto sum{ a + b };
  return sum;
}

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**

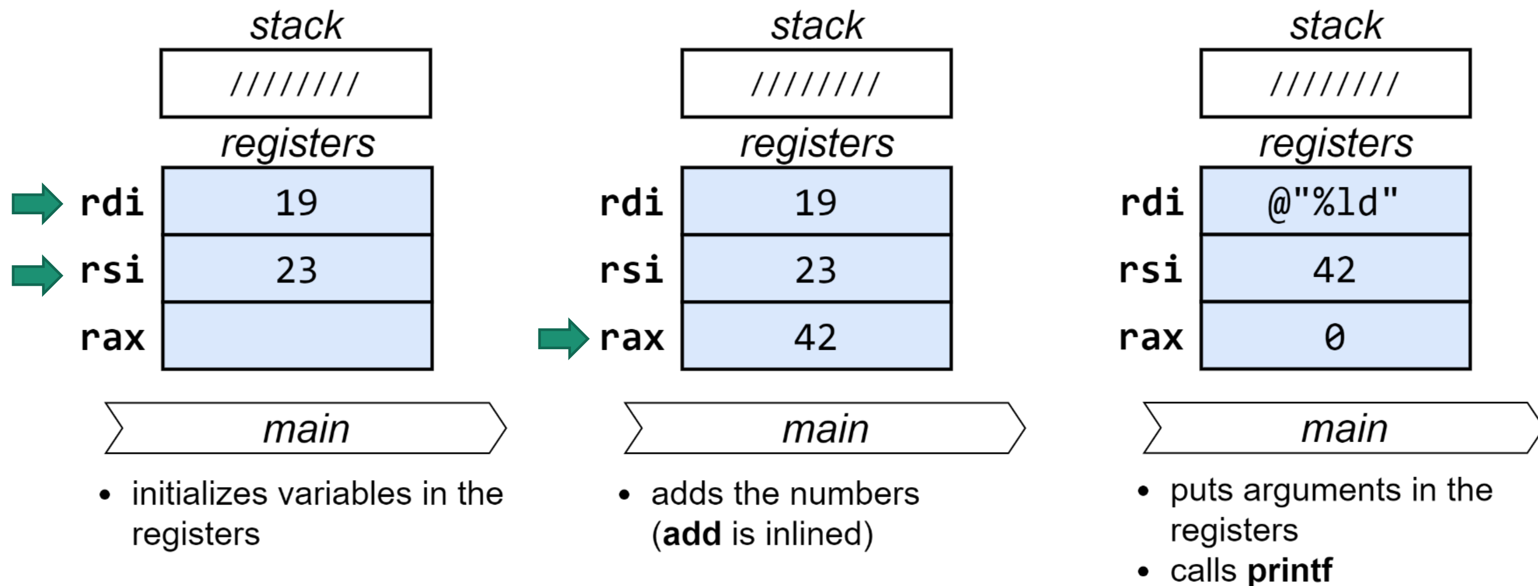
Memory model & ABI 101

```
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

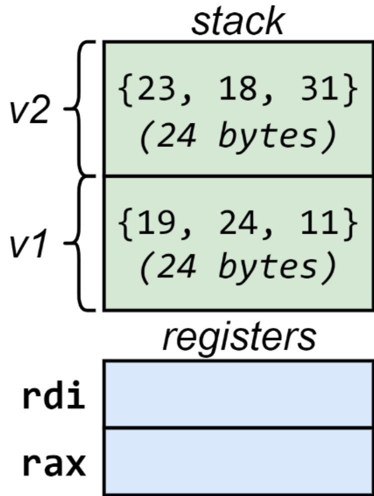
- adds the numbers (**add** is inlined)

- puts arguments in the registers
- calls **printf**

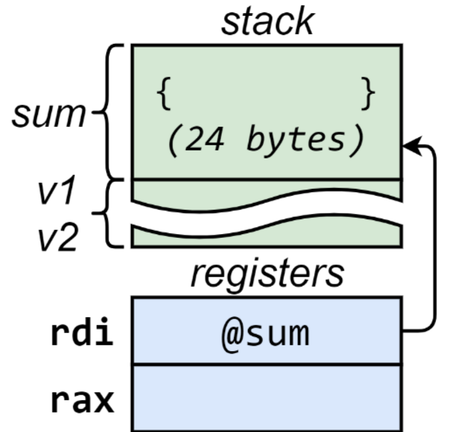
Memory model & ABI 101

```
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);
}
```

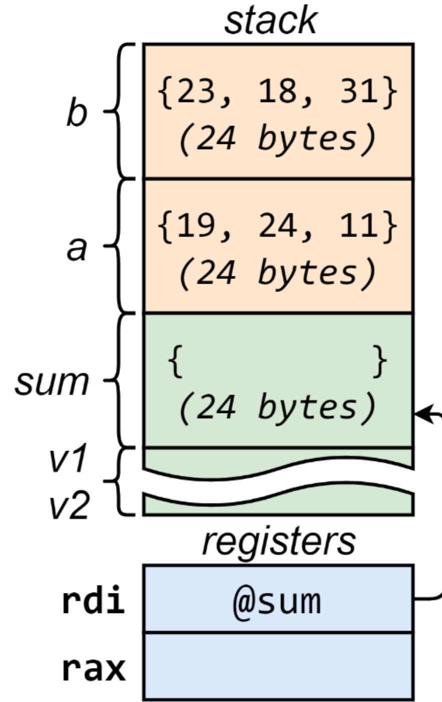
Memory model & ABI 101



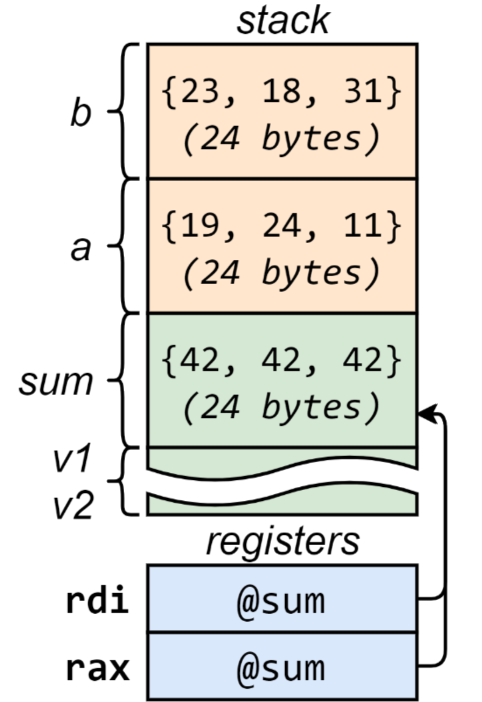
- initializes variables on the stack



- 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**

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, by value (trivial type)

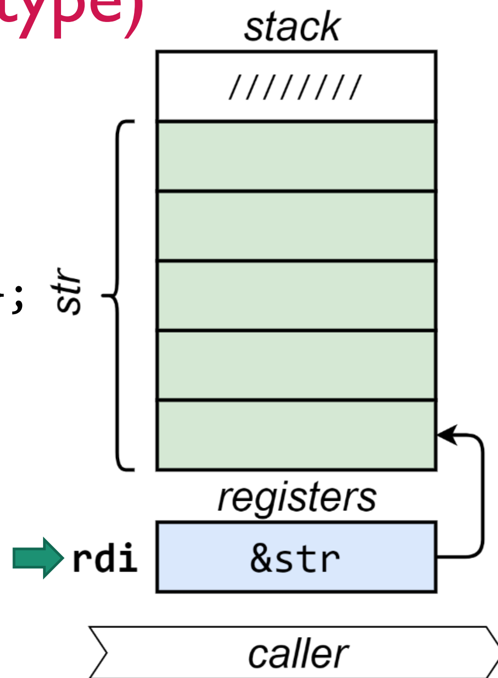
A *trivial* test type:

```
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)

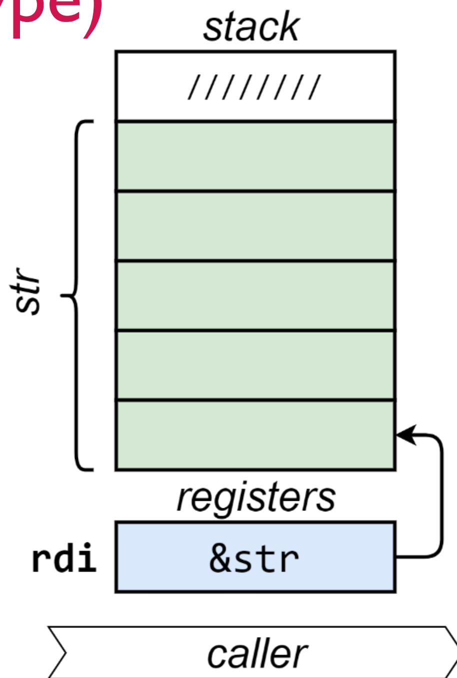
```
trivial_string ret_value(){
  return {22,
          "Alice and Bob love C++"};
}

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



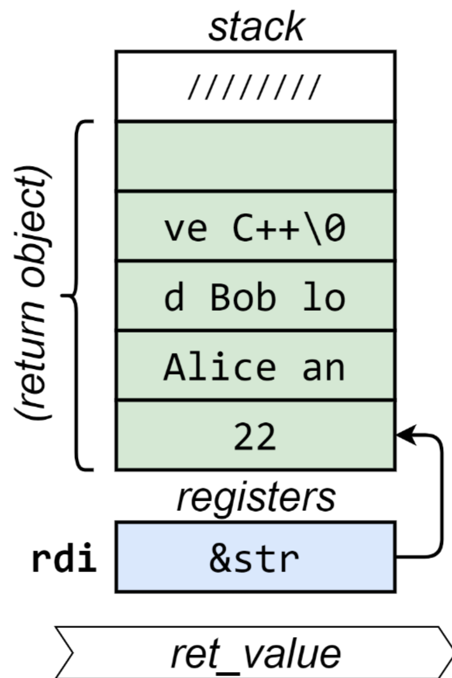
- prepares the stack
- passes the target location of an object to be constrcuted

Returning, by value (trivial type)

```
trivial_string ret_value(){
  return {22,
          "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)

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

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

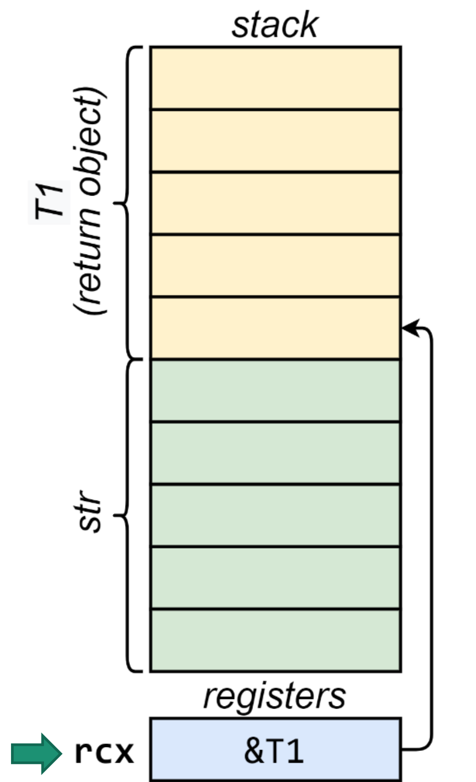
All compilers (gcc, clang, icc) agree

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

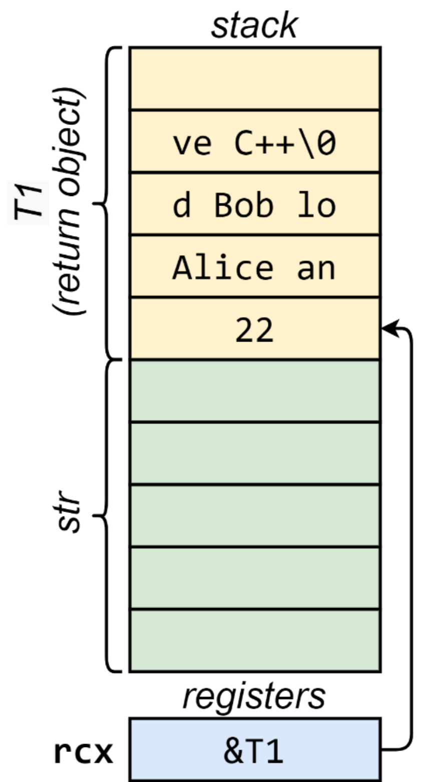
Returning, by value (trivial type)

All compilers besides msvc…

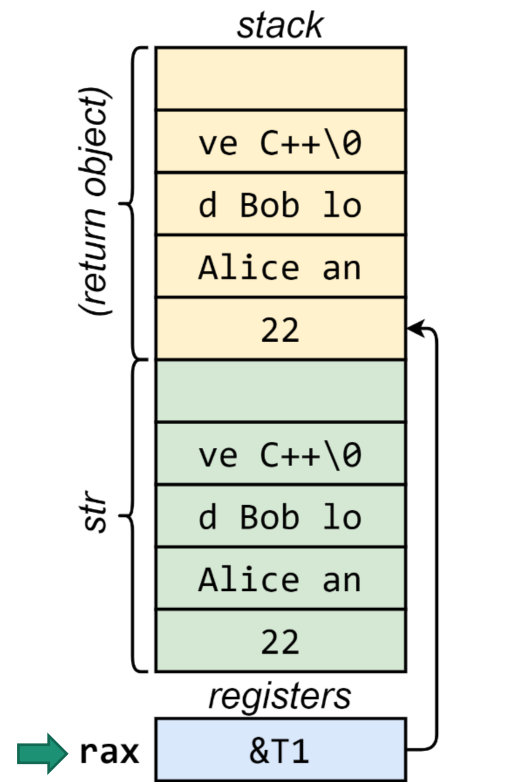
… really



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



- creates an object at the target location



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

Returning, by value – copy elision

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Trivial	✓	✓	✓	&!

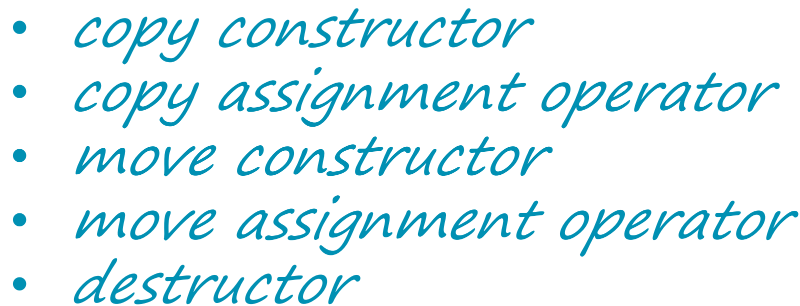
✓ – full copy/ move elision.

& – one copy-construction after callee returns.

Returning non-trivial objects

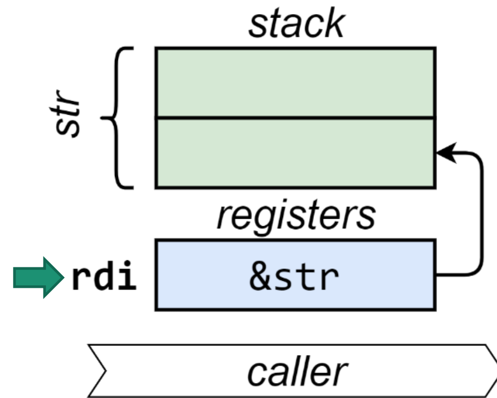
A *non-trivial* test type:

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


  /* + full rule of five */

  const char* c_str() const;

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



Returning, by value (non-trivial type)

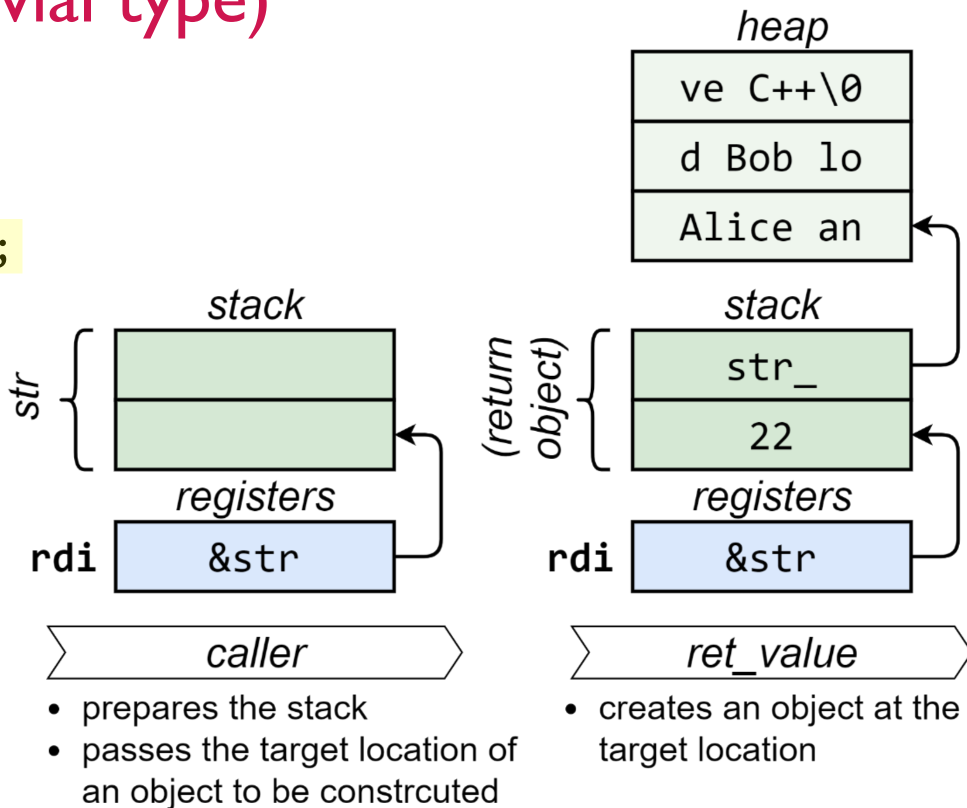
```
proper_string ret_value(){
  return {"Alice and Bob love C++"};
}

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());
```



- Full copy elision on all compilers.
  (really)

Returning, by value – copy elision

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

✓ – full copy/ move elision.

& – one copy-construction after callee returns.

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

But that's only possible when copy ctor/ move ctor exist.

Returning – delayed temporary materialization

C++17

```
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

This is possible because of **delayed temporary materialization** introduced in C++17.

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

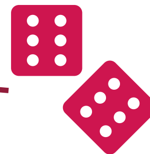
Returning, by value – copy elision

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 (trivial)

```
trivial_string ret_value(){
  trivial_string result{22, "Alice and Bob love C++"};
  if (std::rand() % 2 == 0){
    result = {21, "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;
{
    /* ~~~ */
}
```

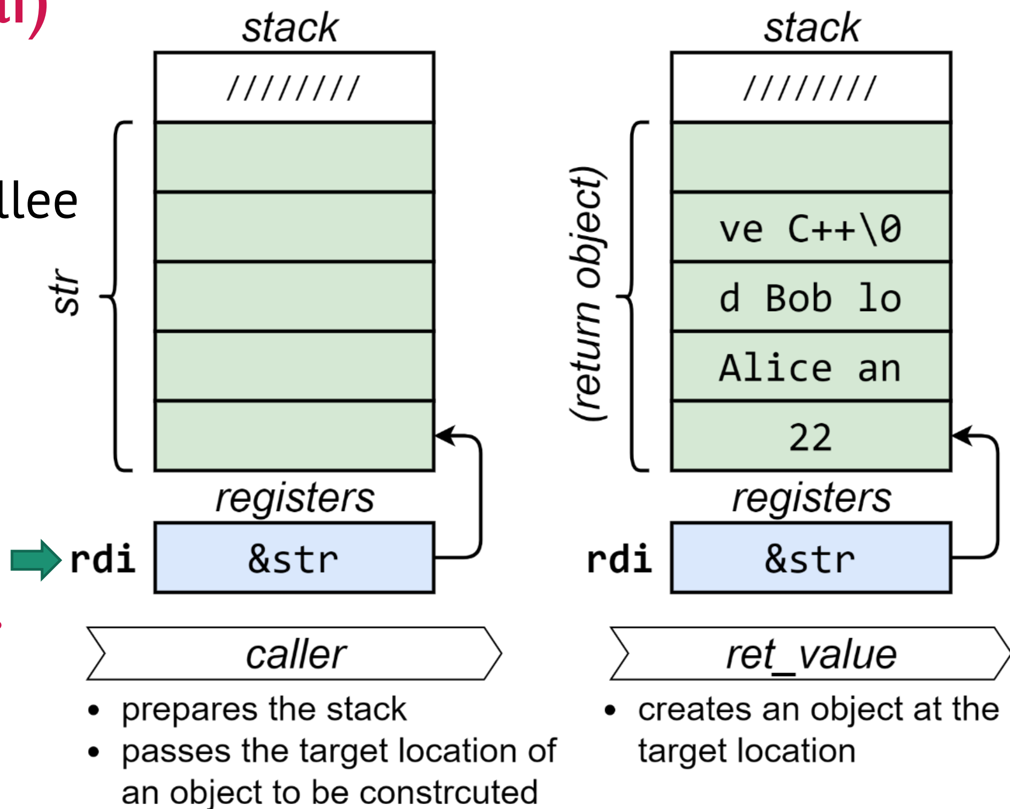
Objects as return values, Michael Tiemann in C++ *Gems* (1998)

Returning, by value II (trivial)

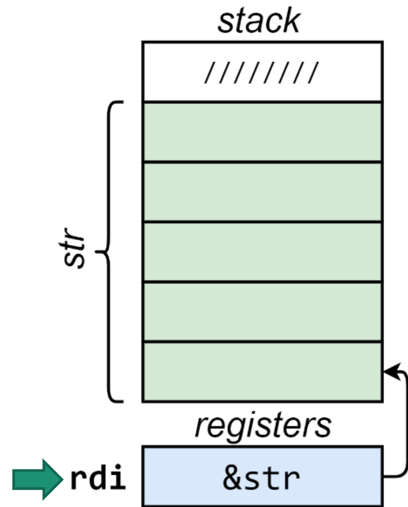
clang & gcc agree:

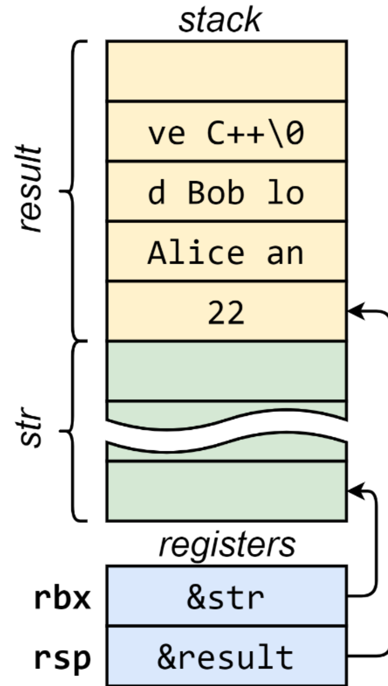
- Object (`result`) created by the callee directly on the stack at the target location
- Full copy elision

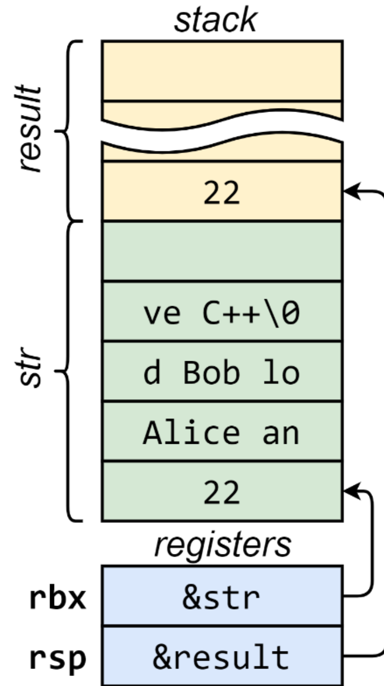
icc & msvc do something else…



Returning, by value II (trivial), icc







caller

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

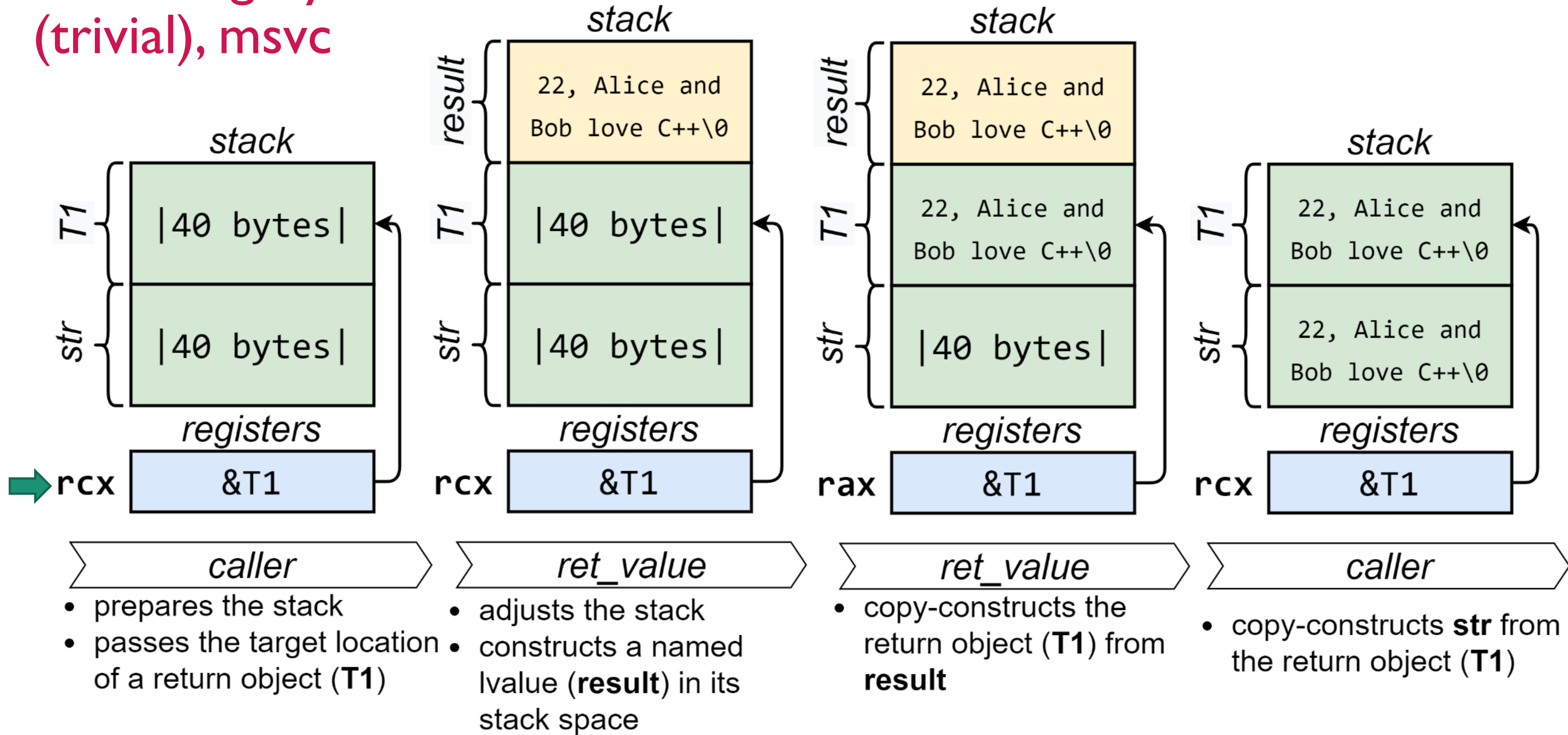
ret_value

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

ret_value

- copies the object to its target destination

Returning, by value II (trivial), msvc



Returning, by value – copy elision

<table>
<tr><th>When</th><th>T</th><th>gcc</th><th>clang</th><th>icc</th><th>msvc</th></tr>
<tr><td rowspan="2"><pre>T function(){
  return T{};
}</pre></td><td>Trivial</td><td>✓</td><td>✓</td><td>✓</td><td>&!</td></tr>
<tr><td>Non-trivial</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr>
<tr><td><pre>T function(){
  T obj{};
  return obj;
}</pre></td><td>Trivial</td><td>✓</td><td>✓</td><td>&</td><td>C&</td></tr>
</table>

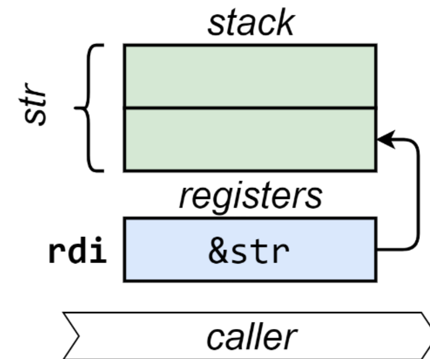
✓ – full copy/ move elision.
C – return object copy-constructed from named lvalue by callee.
& – one copy-construction before (icc)/ after (msvc) callee returns.

Returning, by value II (non-trivial)

```
proper_string ret_value(){
  proper_string result{"Alice and Bob love C++"};
  if (std::rand() % 2 == 0){
    result = "Alice and Bob like C!";
    return result;
  }
  return result;
}

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

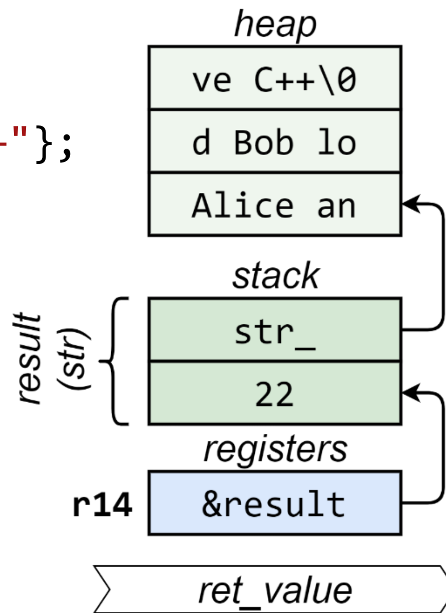
Returning, by value II (non-trivial)

```
proper_string ret_value(){
  proper_string result{"Alice and Bob love C++"};
  if (std::rand() % 2 == 0){
    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)

```
proper_string ret_value(){
  proper_string result{"Alice and Bob love C++"};
  if (std::rand() % 2 == 0){
    result = "Alice and Bob like C!";
    return result;
  }
  return result;
}

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

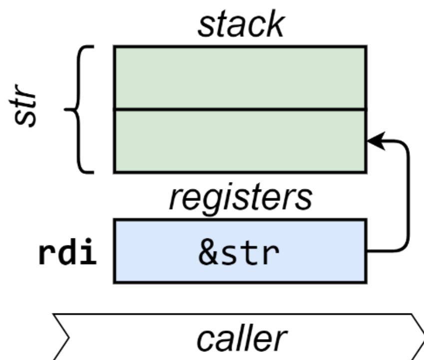


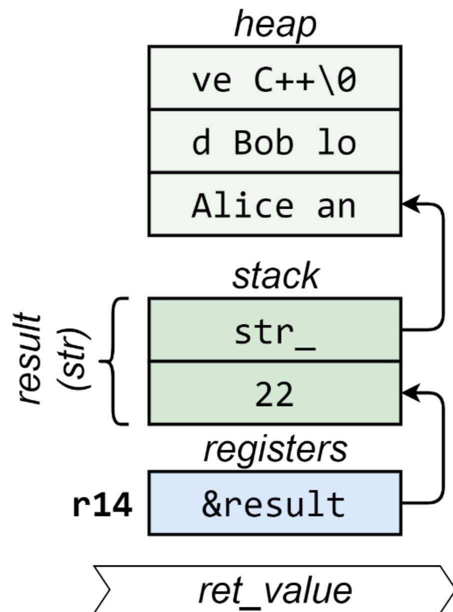
- creates an object at the target location

Returning, by value II (non-trivial)

clang, gcc & icc agree: copy elision

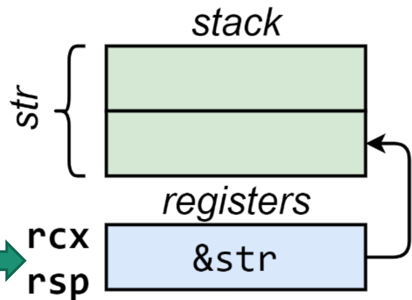
msvc does something else...

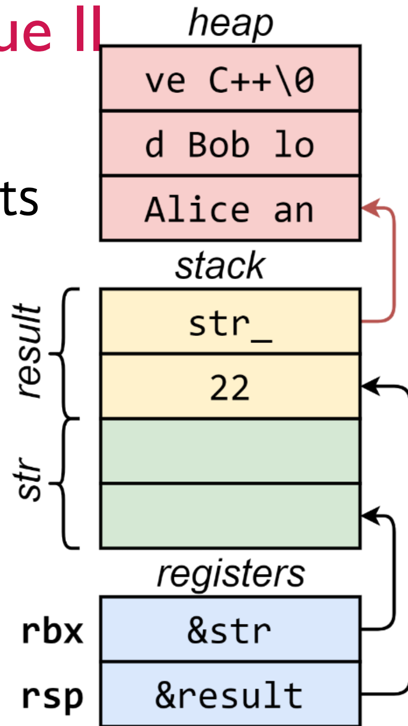


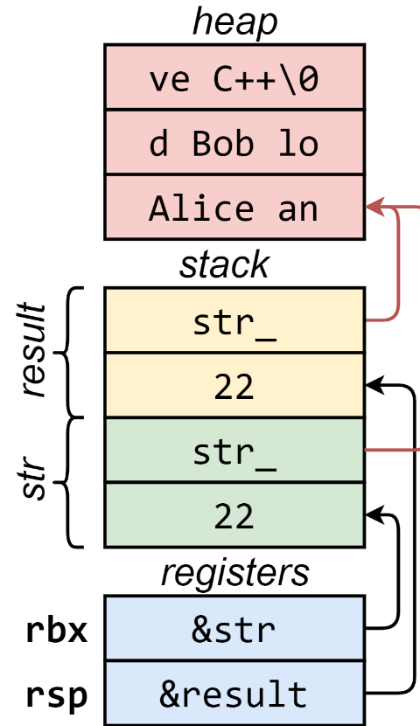


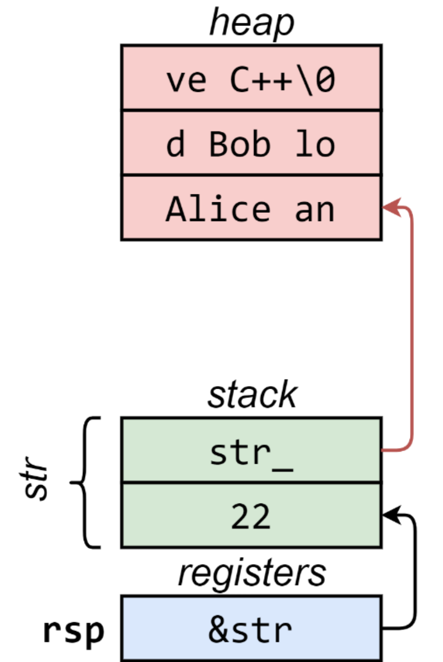
Returning, by value II (non-trivial)

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{}; }`	Trivial	✓	✓	✓	&!
	Non-trivial	✓	✓	✓	✓
`T function(){ T obj{}; return obj; }`	Trivial	✓	✓	&	C&
	Non-trivial	✓	✓	✓	M

✓ – full copy/ move elision.
C – return object copy-constructed from named lvalue by callee.
& – one copy-construction before (icc)/ after (msvc) callee returns.
M – return object move-constructed from named lvalue by callee.

Returning, by value – copy elision

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Move == Copy	✓	✓	✓	✕ !
	Cheaply movable	✓	✓	✓	✓
`T function(){ T obj{}; return obj; }`	Move == Copy	✓	✓	✕	✕ ✕
	Cheaply movable	✓	✓	✓	⚡

✓ – full copy/ move elision.
✕ – object is copied.
⚡ – object is moved.

Let's step up the game

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{}; }`	Move == Copy	=&	=&	=&	=&
	Cheaply movable	=&&	=&&	=&&	=&&
`T function(){ T obj{}; return obj; }`	Move == Copy	=&	=&	=&	C=&
	Cheaply movable	=&&	=&&	=&&	M=&&

C – return object copy-constructed (M – move) from named lvalue.
=& – copy-assignment after/ before (icc) callee returns.
=&& – move-assignment after callee returns.

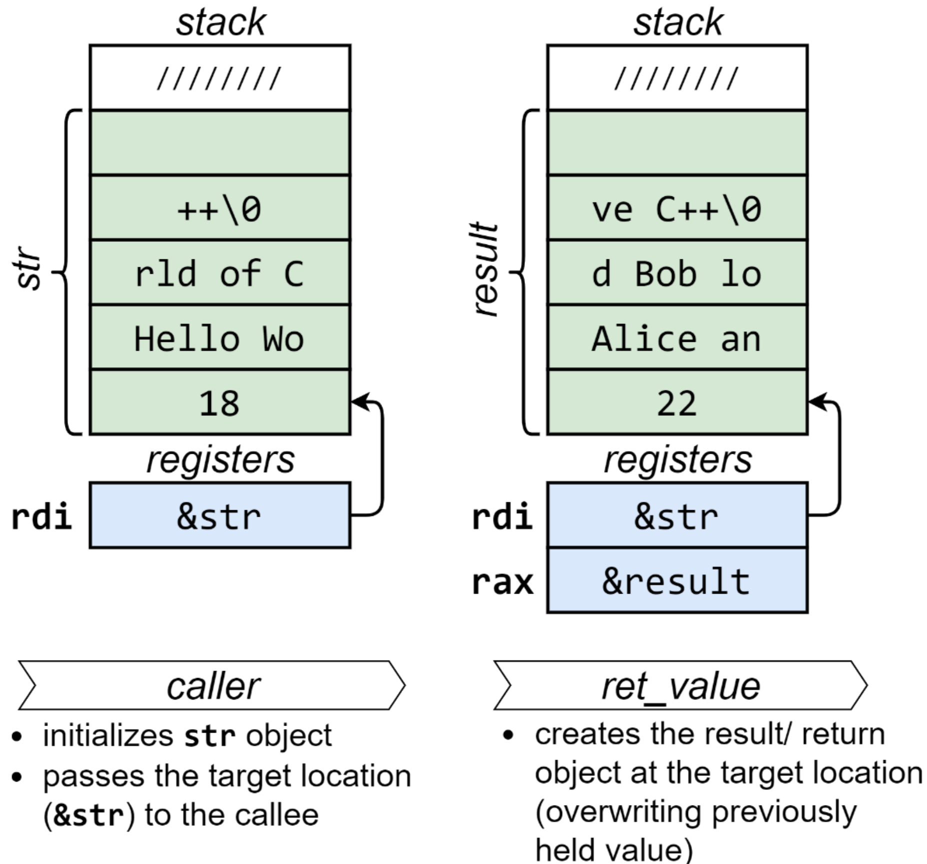
Returning, by value – copy elision (assignment)

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Move == Copy	✓	=&	✓	=&
	Cheaply movable	✓ / =&&	✓ /=&&	=&&	=&&
`T function(){ T obj{}; return obj; }`	Move == Copy	=&	=&	=&	C=&
	Cheaply movable	✓/ =&&	✓ /=&&	=&&	M=&&

✓ – full copy/ move elision (as unbelievable as it sounds).
C – return object copy-constructed (M – move) from named lvalue.
=& – copy-assignment after/ before (icc) callee returns.
=&& – move-assignment after callee returns.

Returning, by value, gcc & 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());
```



Returning, by value – copy elision (assignment)

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Move == Copy	✓	=&	✓	=&
	Cheaply movable	✓ / =&&	✓ /=&&	=&&	=&&
`T function(){ T obj{}; return obj; }`	Move == Copy	=&	=&	=&	C=&
	Cheaply movable	✓/ =&&	✓ /=&&	=&&	M=&&

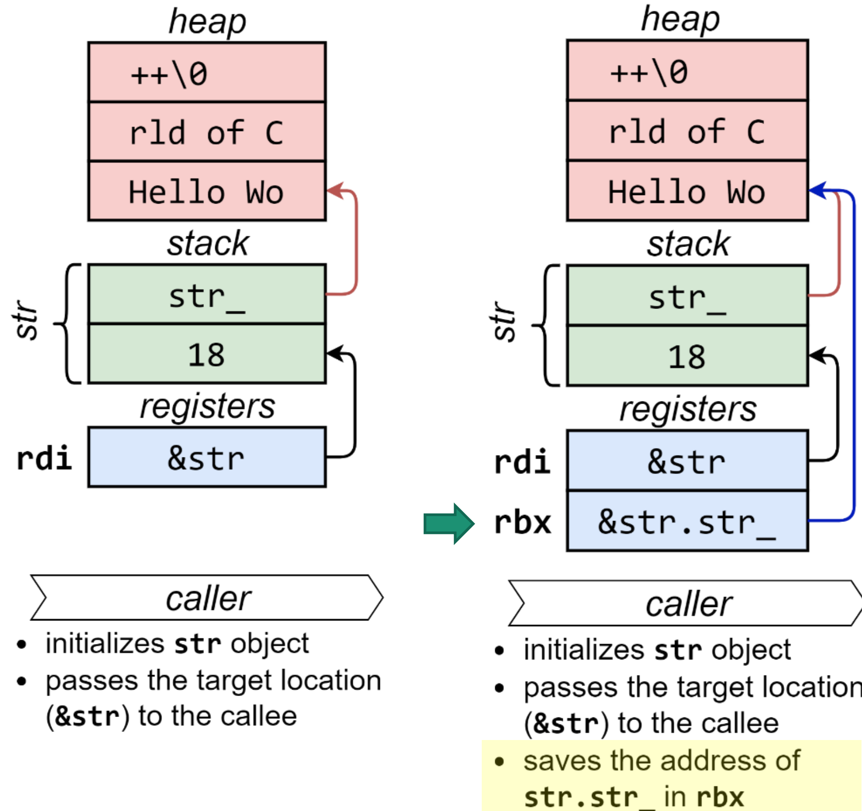
✓ – full copy/ move elision (as unbelievable as it sounds).
C – return object copy-constructed (M – move) from named lvalue.
=& – copy-assignment after/ before (icc) callee returns.
=&& – move-assignment after callee returns.

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

```
proper_string ret_value(){
  proper_string result{"Alice and Bob love C++"};
  if (std::rand() % 2 == 0){
  /* ~~~ */
}
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



heap

++\0

rld of C

Hello Wo

stack

str

str_

18

registers

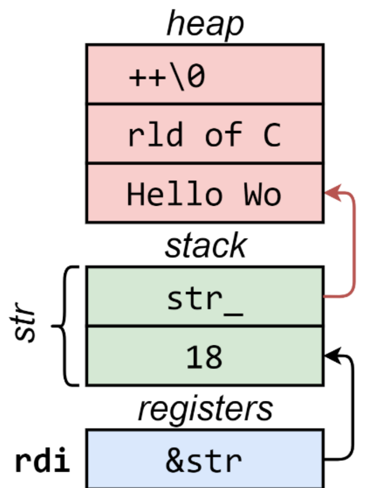
rdi &str

rbx &str.str_

caller

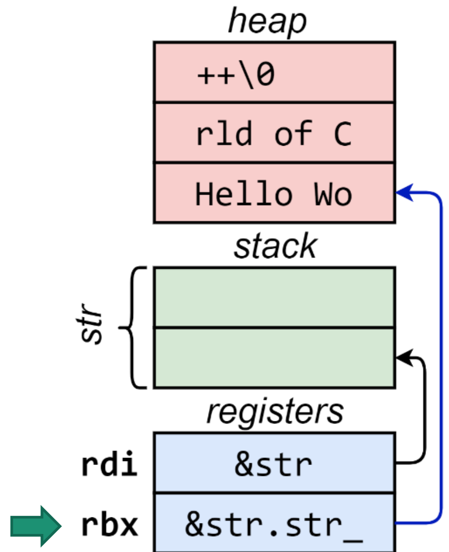
- initializes **str** object
- passes the target location (**&str**) to the callee
- saves the address of **str.str_** in **rbx**

gcc, clang:
non-trivial, lvalue & assignment



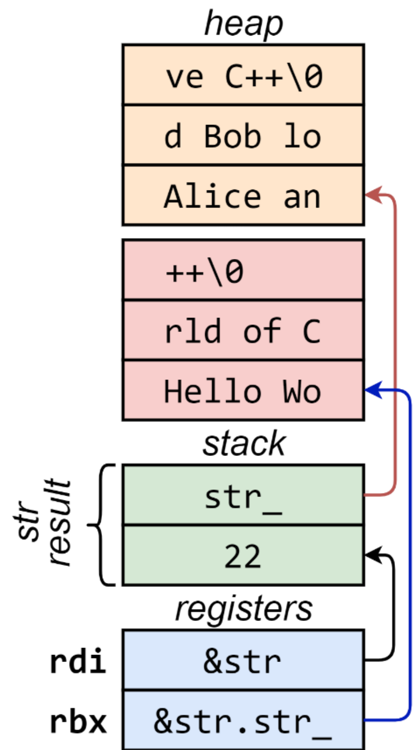
caller

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



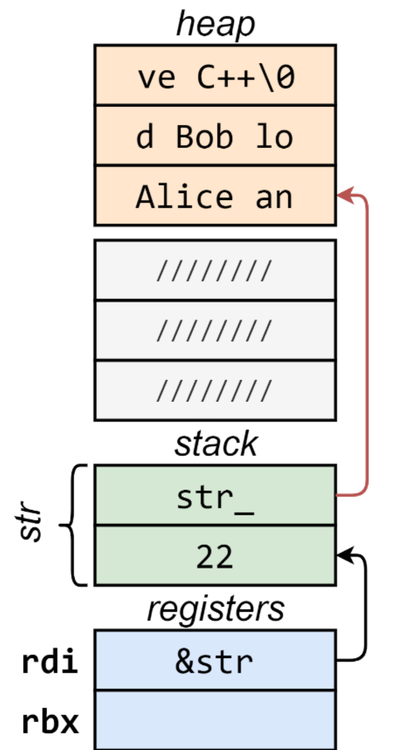
caller

- 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)



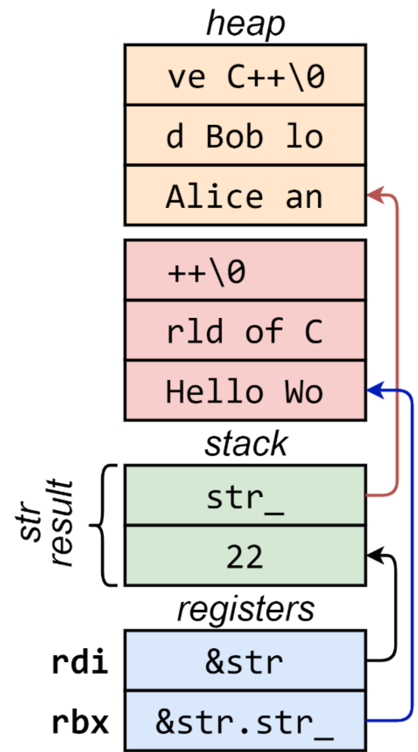
caller

- deallocates memory held in **rbx**

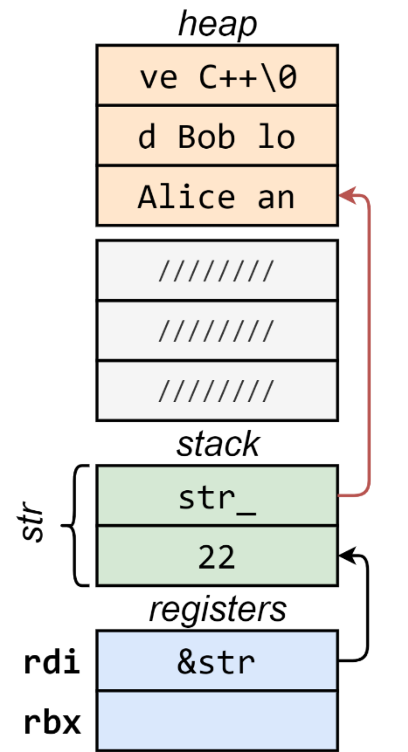
```
proper_string&
operator=(proper_string&& other){
  if (this != &other){
    ::operator delete(str_);
    len_ = other.len_;
    str_ = other.str_;
    other.len_ = 0;
    other.str_ = nullptr;
  }
  return *this;
}

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

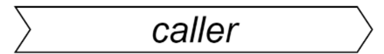
caller

ret_val



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





- deallocates memory held in **rbx**

Returning, by value – copy elision

When	T	gcc	clang	icc	msvc
`T function(){ return T{}; }`	Non-movable	✓	✕	✓	✕
	Movable	✓/⚡	✓/⚡	⚡	⚡
`T function(){ T obj{}; return obj; }`	Non-movable	✕	✕	✕	✕ ✕
	Movable	✓/⚡	✓/⚡	⚡	⚡⚡

✓ – full copy/ move elision.
✕ – object is copied.
⚡ – object is moved.

Copy/ move elision is guaranteed (since C++17) for prvalues.

Sometimes copy/ move elision works for lvalues.

Surprisingly, even whole chains of copies/ moves can be eliminated.

Passing, by value

Passing, by value

Pass by-value usually means:

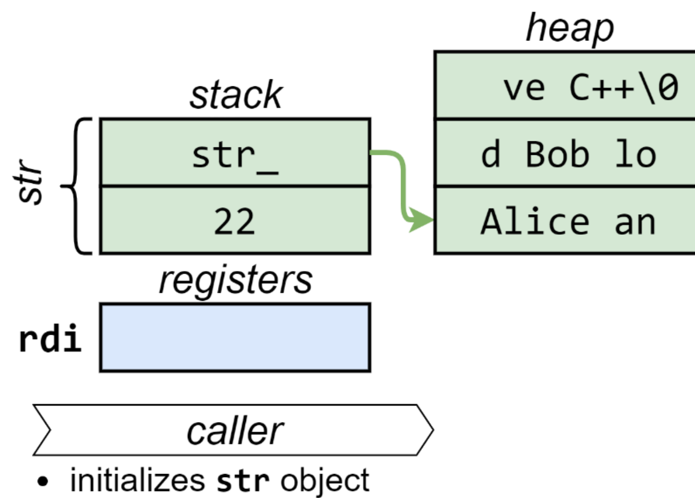
```
void function(SomeType arg){
    /* ~~~ */
}

/* ~~~ */

SomeType obj{};
function(obj);
```

Passing, by value, non-trivial

```
void by_value(proper_string arg){
  printf("%s", arg.c_str());
}
```

```
proper_string str{
      "Alice and Bob love C++"};
```

```
by_value(str);
```

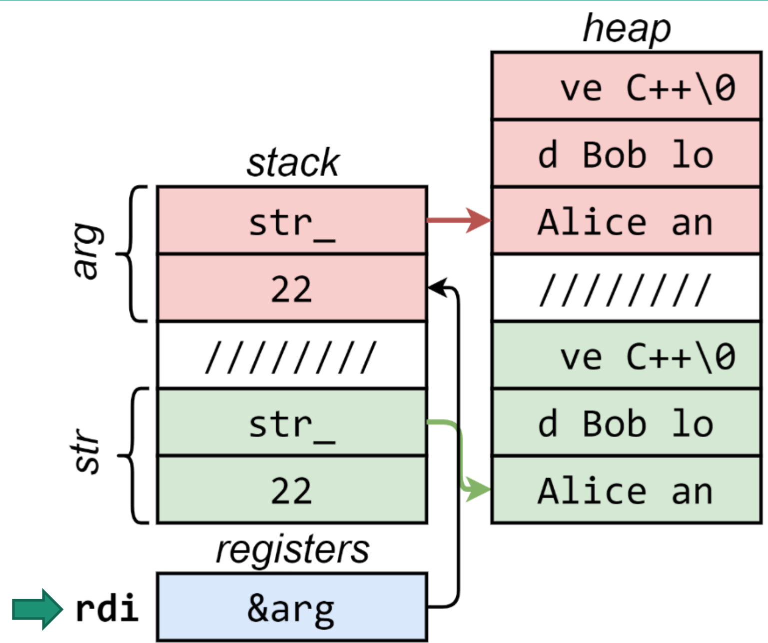


Passing, by value, non-trivial

```
void by_value(proper_string arg){
  printf("%s", arg.c_str());
}

proper_string str{
     "Alice and Bob love C++"};

by_value(str);
```

All compilers fully to agree:

- ctor + copy-ctor
- 2x ::operator new
  2x ::operator delete

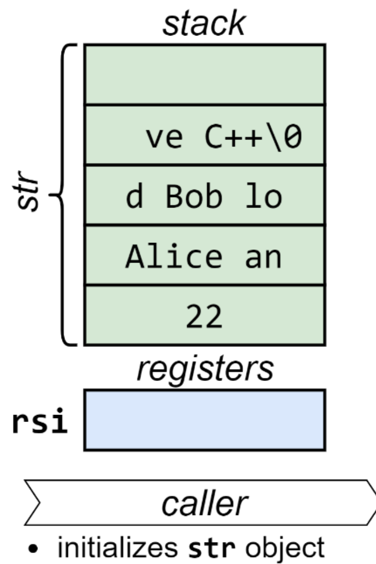


caller

- makes a full copy of **str** (**arg**)
- passes location of argument (**&arg**) to the callee

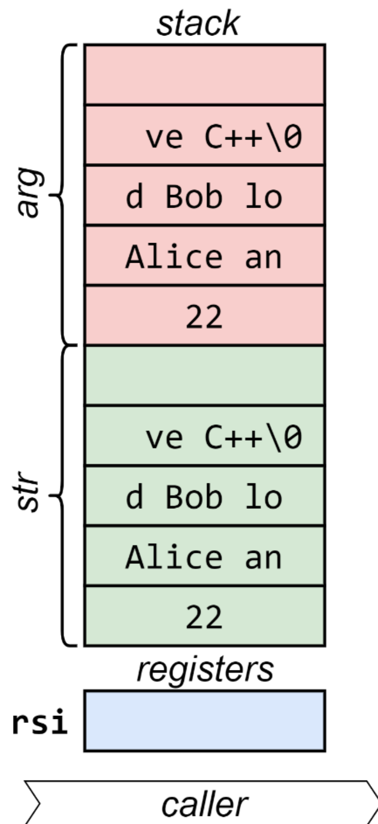
Passing, by value, trivial

```
void by_value(trivial_string arg){
  printf("%s", arg.c_str());
}

trivial_string str{22,
     "Alice and Bob love C++"};

by_value(str);
```



Passing, by value, trivial

```
void by_value(trivial_string arg){
  printf("%s", arg.c_str());
}

trivial_string str{22,
      "Alice and Bob love C++"};

by_value(str);
```

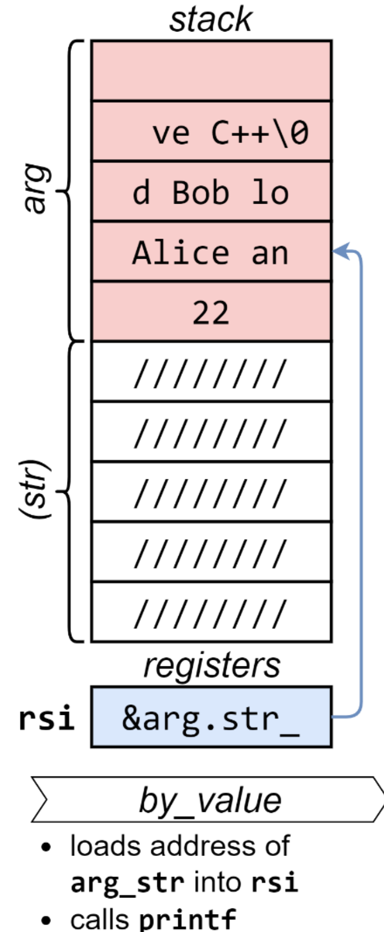


- makes a full copy of **str** (**arg**)
- calls **by_value**

Passing, by value, trivial

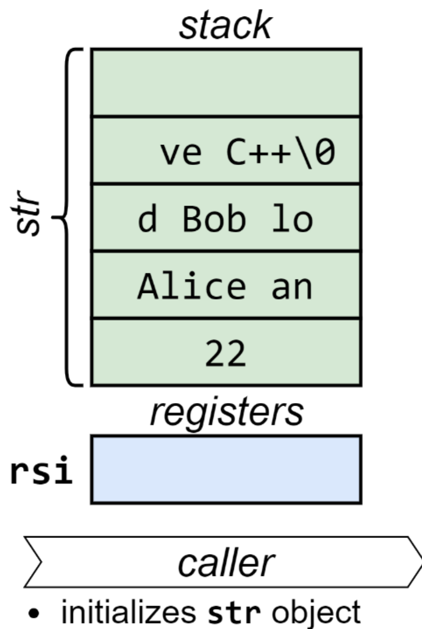
```
void by_value(trivial_string arg){
  printf("%s", arg.c_str());
}

trivial_string str{22,
     "Alice and Bob love C++"};

by_value(str);
```

- That's what `gcc` and `msvc` do.
- `clang` and `icc` take a different approach.



Passing, by value, trivial (clang & icc way)

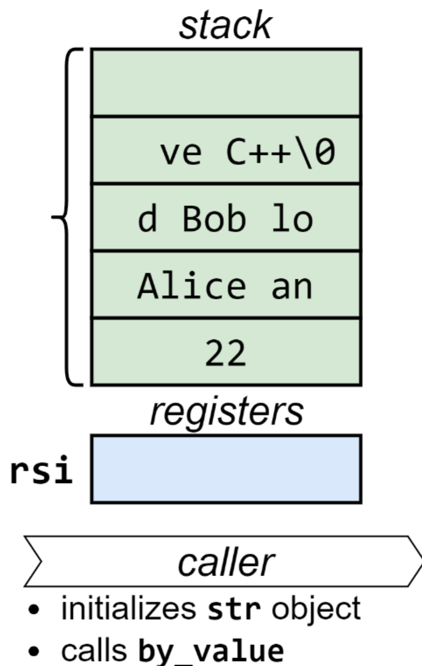
```
void by_value(trivial_string arg){
  printf("%s", arg.c_str());
}

trivial_string str{22,
     "Alice and Bob love C++"};


by_value(str);
```



Passing, by value, trivial (clang & icc way)

```
void by_value(trivial_string arg){
  printf("%s", arg.c_str());
}

trivial_string str{22,
      "Alice and Bob love C++"};
```

```
by_value(str);
```



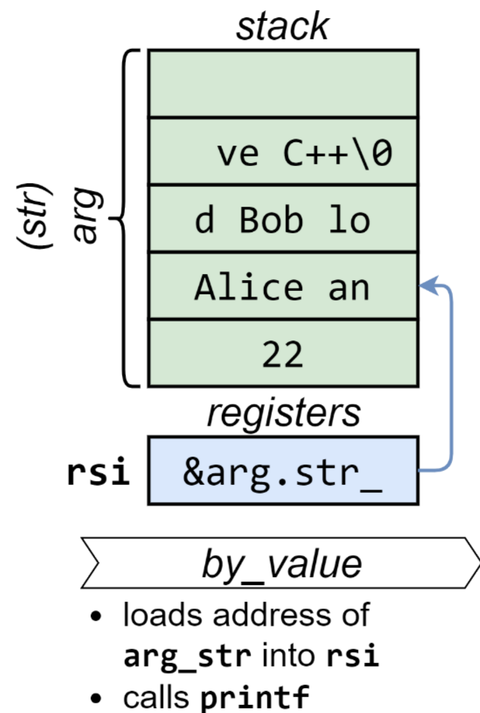
Passing, by value, trivial (clang & icc way)

```
void by_value(trivial_string arg){
  printf("%s", arg.c_str());
}
```

```
trivial_string str{22,
      "Alice and Bob love C++"};
```

```
by_value(str);
```

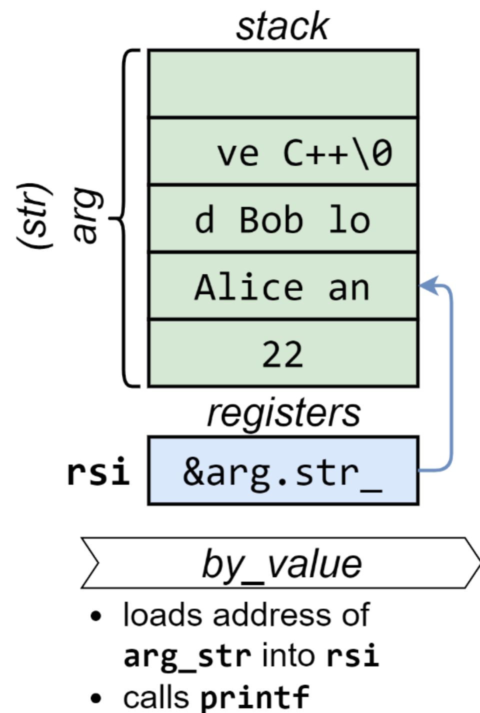
- No copy is made when passing by value.



Passing, by value, trivial (clang & icc way)

```
void by_value(trivial_string arg){
  printf("%s", arg.c_str());
}

trivial_string str{22,
     "Alice and Bob love C++"};

printf("%lu", str.len_);

by_value(str);
```

- No copy is made when passing by value.
- But only if this line is **not present**.



Passing, by value, trivial (clang & icc way)

```
main:
    sub     rsp, 40
    mov     rax, qword ptr [rip + .str+32]
    mov     qword ptr [rsp + 32], rax
    movups  xmm0, xmmword ptr [rip + .str+16]
    movups  xmmword ptr [rsp + 16], xmm0
    movups  xmm0, xmmword ptr [rip + .str]
    movups  xmmword ptr [rsp], xmm0
    call    by_value(trivial_string)
    xor     eax, eax
    add     rsp, 40
    ret
```

*Creating **str** and calling **by_value***

Passing, by value, trivial (clang & icc way)

```
void by_value(trivial_string arg){
  printf("%s", arg.c_str());
}

trivial_string str{22,
      "Alice and Bob love C++"};

printf("%lu", str.len_);

by_value(str);
```

Passing, by value, trivial (clang & icc way)

```
main:
    sub     rsp, 88
    mov     edi, offset .L.str.1    # pass "%lu"
    mov     esi, 22                 # pass 22 (constant)!
    xor     eax, eax
    call    printf                  # call printf

    mov     qword ptr [rsp + 48], 22     # str
    #       ... create str on the stack
    mov     rax, qword ptr [rsp + 80]    # arg
    #       ... create a copy of str (arg)
    call    by_value(trivial_string)
    add     rsp, 88
    xor     eax, eax
    ret
```

Creating str and arg and calling by_value

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)

So many possibilities…

```
void by_val(proper_string str){
    printf("%s", str.c_str());
}
```

```
void by_crref(const proper_string&& str){
    printf("%s", str.c_str());
}
```

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

```
void by_ref(proper_string& str){
    printf("%s", str.c_str());
}
```

```
void by_rref(proper_string&& str){
    printf("%s", str.c_str());
}
```

```
void by_ptr(proper_string* str){
    printf("%s", str->c_str());
}
```

```
void by_cptr(const proper_string* str){
    printf("%s", str.c_str());
}
```

How many different assembly representations?

gcc, clang, icc:

```
by_doesnt_matter(proper_string):
        sub     rsp, 8
        mov     rsi, QWORD PTR [rdi+8]
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        add     rsp, 8
        ret
```

pointer to string's char data

format string "%s"

msvc: by_value is somewhat lenghty…

TIME FOR ANSWERS!

to pass and return -

the story of functions, values and compilers

Dawid Zalewski

github.com/zaldawid

zaldawid@gmail.com

saxion.edu