Parameterizing Behavior

Classical polymorphism versus generic programming

Arthur O’Dwyer

2017-04-19

What is polymorphism?

To answer this question, we must first ask:

What is monomorphism?

Monomorphic code is “unparameterized”

```
std::optional<Player> winner(const Grid&);
Move think_of_a_move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

What are some dimensions of parametrization we could play with?

```
std::optional<Player> winner(const Grid&);
Move think_of_a_move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

What are some dimensions of parametrization we could play with?

```
std::optional<Player> winner(const Grid&);
Move think_of_a_move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- AI Strategy: we could change the behavior of “think_of_a_move” to be smarter

What are some dimensions of parametrization we could play with?

```
std::optional<Player> winner(const Grid&);
Move think_of_a_move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- AI Strategy: we could change the behavior of “think_of_a_move” to be smarter
- Winner: we could change the behavior of “winner”, for example to let the first player win cat games (that is, ties)

What are some dimensions of parametrization we could play with?

```
std::optional<Player> winner(const Grid&);
Move think_of_a_move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- AI Strategy: we could change the behavior of “think_of_a_move” to be smarter
- Winner: we could change the behavior of “winner”, for example to let the first player win cat games (that is, ties)
- We could play a different game entirely — e.g., Checkers instead of Tic-Tac-Toe.

What are some dimensions of parametrization we could play with?

```
std::optional<Player> winner(const Grid&);
Move think_of_a_move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- AI Strategy: we could change the behavior of “think_of_a_move” to be smarter
- Winner: we could change the behavior of “winner”, for example to let the first player win cat games (that is, ties)
- We could play a different game entirely — e.g., Checkers instead of Tic-Tac-Toe.
- Instead of just evaluating one possible move, we could evaluate *all* possible moves at every step.

What are some dimensions of parametrization we could play with?

```
std::optional<Player> winner(const Grid&);
Move think_of_a_move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- AI Strategy: we could change the behavior of "think_of_a_move" to be smarter
- Winner: we could change the behavior of "winner", for example to let the first player win cat games (that is, ties)
- We could play a different game entirely — e.g., Checkers instead of Tic-Tac-Toe.
- Instead of just evaluating one possible move, we could evaluate *all* possible moves at every step.
- Instead of evaluating moves in a game at all, we could implement an enterprise relational database.

What are some dimensions of parametrization we could play with?

```
std::optional<Player> winner(const Grid&);
Move think_of_a_move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- AI Strategy: we could change the behavior of “think_of_a_move” to be smarter
- Winner: we could change the behavior of “winner”, for example to let the first player win cat games (that is, ties)
- We could play a different game entirely — e.g., Checkers instead of Tic-Tac-Toe.
- Instead of just evaluating one possible move, we could evaluate *all* possible moves at every step.
- Instead of evaluating moves in a game at all, we could implement an enterprise relational database.
- Instead of doing computer programming, we could become crab fishermen.

Let’s look at the assembly code.

```
__Z33play_tic_tac_toe_against_yourself4Grid:
     pushq %r15
     pushq %r14
     pushq %r12
     pushq %rbx
     subq  $24, %rsp
     movq  %rdi, %r14
     leaq  64(%rsp), %rbx
     leaq  16(%rsp), %r15
     leaq  8(%rsp), %r12
     jmp   LBB0_1
LBB0_2:
     movq  %rbx, %rdi
     callq __Z15think_of_a_moveRK4Grid
     movq  %rax, 8(%rsp)
     movq  %rbx, %rdi
     movq  %r12, %rsi
     callq __ZpLR4GridRK4Move
LBB0_1:
     movq  %r15, %rdi
     movq  %rbx, %rsi
     callq __Z6winnerRK4Grid
     cmpb  $0, 16(%rsp)
     je    LBB0_2
     movb  $0, 16(%rsp)
     movq  %r14, %rdi
     movq  %rbx, %rsi
     callq __Z6winnerRK4Grid
     movq  %r14, %rax
     addq  $24, %rsp
     popq  %rbx
     popq  %r12
     popq  %r14
     popq  %r15
     retq
```

Let’s look at the assembly code.

```
__Z33play_tic_tac_toe_against_yourself4Grid:
        pushq %r15
        pushq %r14
        pushq %r12
        pushq %rbx
        subq  $24, %rsp
        movq  %rdi, %r14
        leaq  64(%rsp), %rbx
        leaq  16(%rsp), %r15
        leaq  8(%rsp), %r12
        jmp   LBB0_1
LBB0_2:
        movq  %rbx, %rdi
        callq __Z15think_of_a_moveRK4Grid
        movq  %rax, 8(%rsp)
        movq  %rbx, %rdi
        movq  %r12, %rsi
        callq __ZpLR4GridRK4Move  # operator+=
LBB0_1:
        movq  %r15, %rdi
        movq  %rbx, %rsi
        callq __Z6winnerRK4Grid
        cmpb  $0, 16(%rsp)
        je    LBB0_2
        movb  $0, 16(%rsp)
        movq  %r14, %rdi
        movq  %rbx, %rsi
        callq __Z6winnerRK4Grid
        movq  %r14, %rax
        addq  $24, %rsp
        popq  %rbx
        popq  %r12
        popq  %r14
        popq  %r15
        retq
```

Let’s look at the assembly code.

Text section

```
play_tic_tac_toe_against_yourself:
...%rbx, %rdi; callq __Z15think_of_a_moveRK4Grid; movq  %rax, 8(%rsp); ...

think_of_a_move:
...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; movq %rdi,
...
```

Here the address of `think_of_a_move` is 100% hard-coded into the `play_tic_tac_toe_against_yourself` function.

STL things that work like this: `operator new()`

We could use a function pointer.

```
std::optional<Player> winner(const Grid&);
Move (*think_of_a_move)(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

We could use a function pointer.

Text section

`play_tic_tac_toe_against_yourself`:

```
...%rbx, %rdi; callq *think_of_a_move(%rip); movq  %rax, 8(%rsp); ...
```

Here the address of `think_of_a_move` is still hard-coded,
but we can reseat that pointer to change the function's behavior.

```
...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; movq %rdi, ...
```

Data section

`think_of_a_move`: pointer

STL things that work like this: `set_terminate()`

We could use a function pointer.

Text section

`play_tic_tac_toe_against_yourself`:

```
...%rbx, %rdi; callq *%fs:think_of_a_move@tpoff; movq  %rax, 8(%rsp); ...
```

Here the address of `think_of_a_move` is still hard-coded,
but we can reseat that pointer to change the function's behavior.

```
...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; movq %rdi, ...
```

***Thread-local** data section*

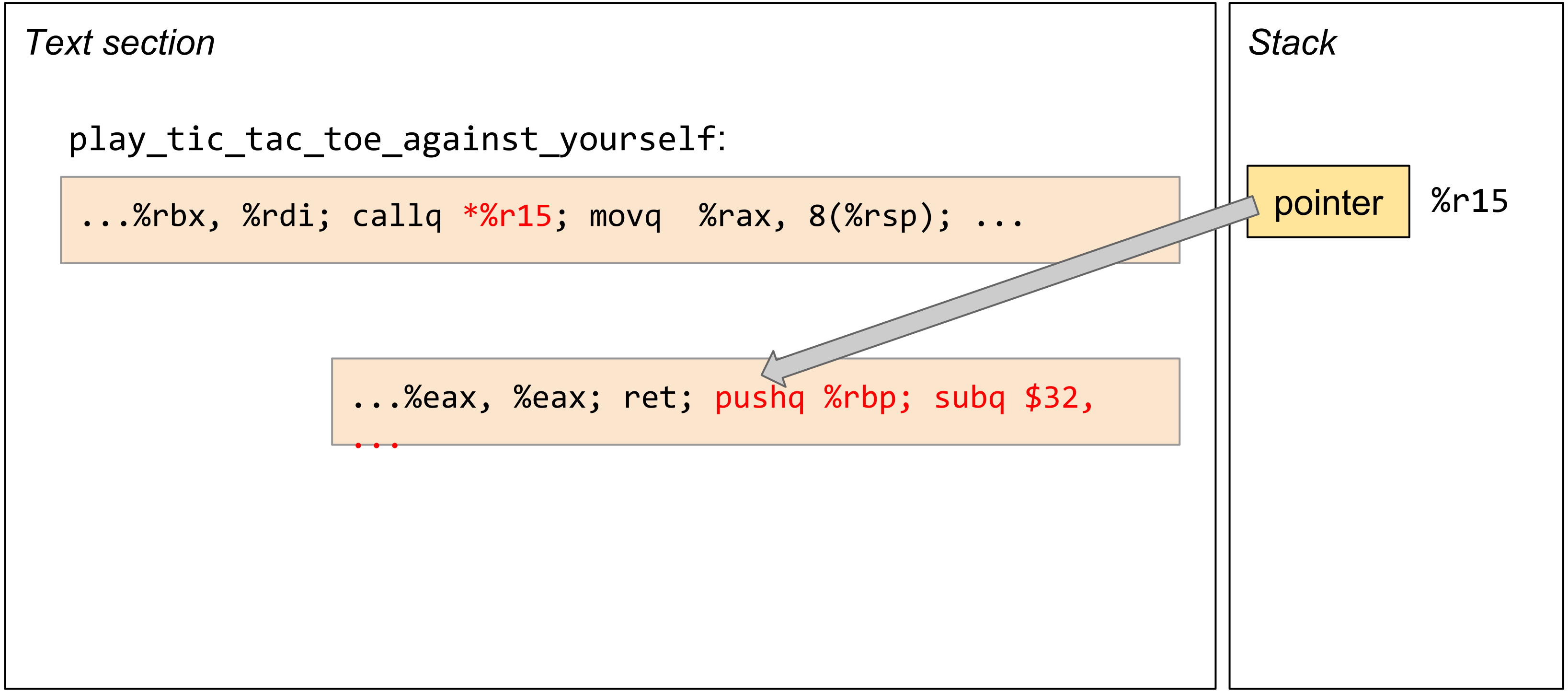
`think_of_a_move`: pointer

STL things that work like this: `set_terminate()`

We could *pass in* a function pointer.

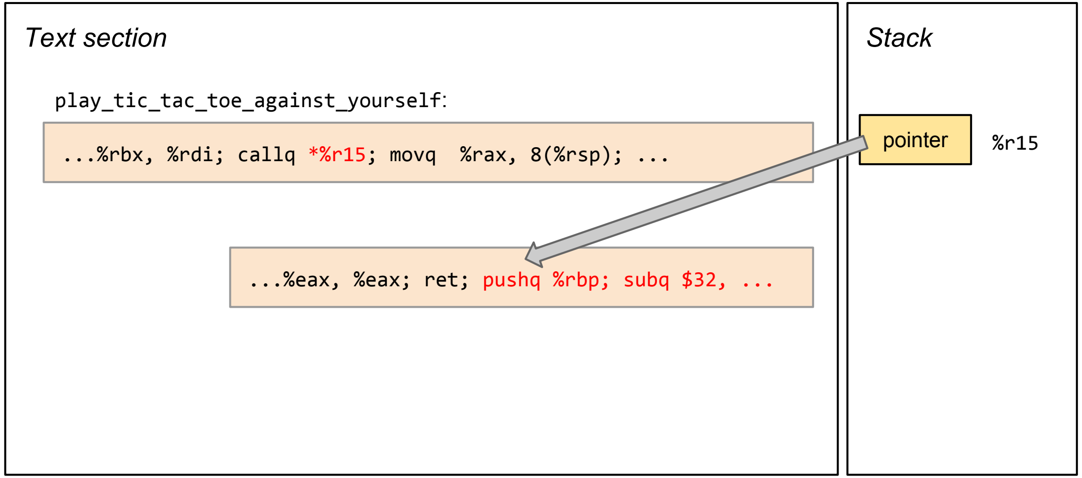
```
std::optional<Player> winner(const Grid&);
using TOAM_t = Move(const Grid&);
Grid& operator+=(Grid&, const Move&);

auto play_tic_tac_toe_against_yourself(Grid position, TOAM_t *think_of_a_move)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

We could *pass in* a function pointer.

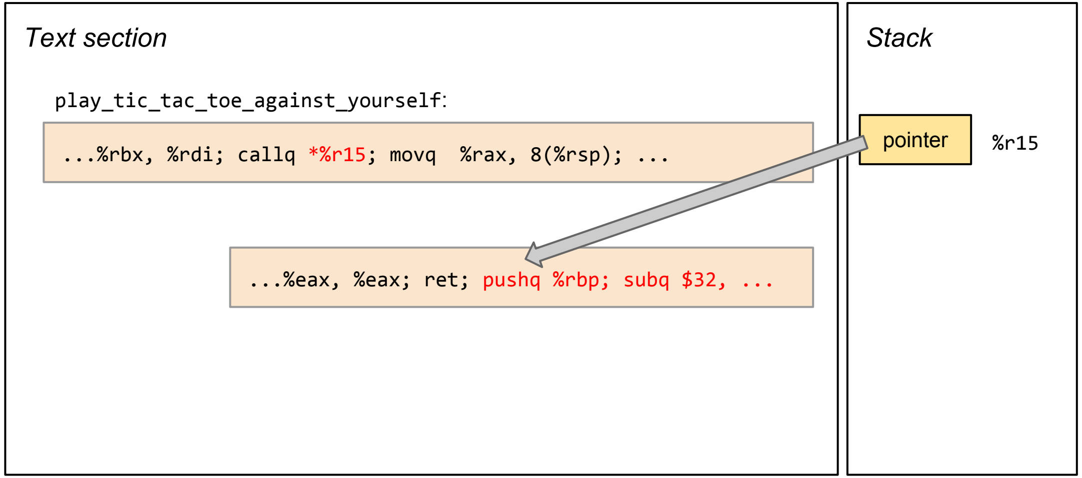


STL things that work like this: `qsort()`, `bsearch()`

Problems with this approach?

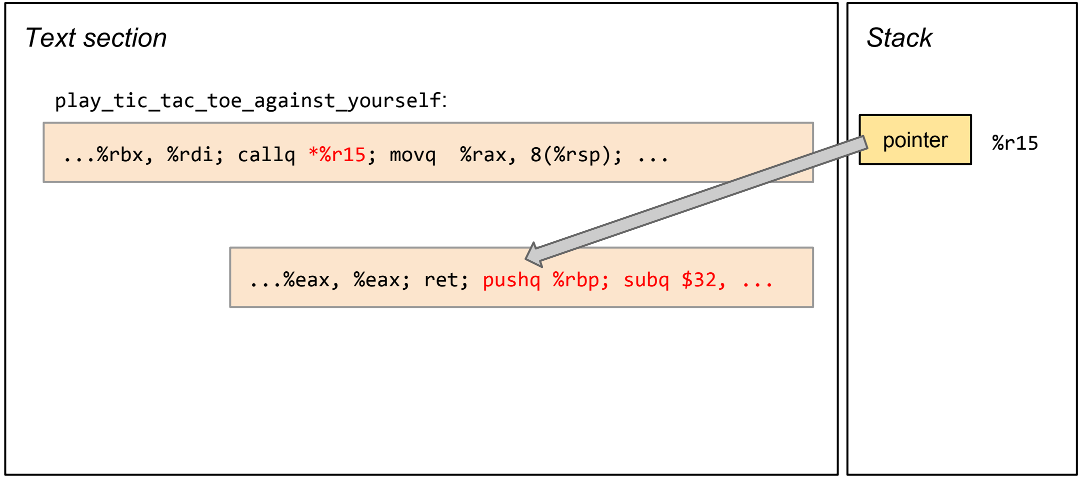


Problems with this approach?



- What if the callback function `think_of_a_move` needs some extra information? That is, what if ***its*** behavior is parametrized by some “adverb”?
- We could pass in a “cookie” containing that adverbial information. (STL things that work like this: `qsort_r()`.)

Problems with this approach?

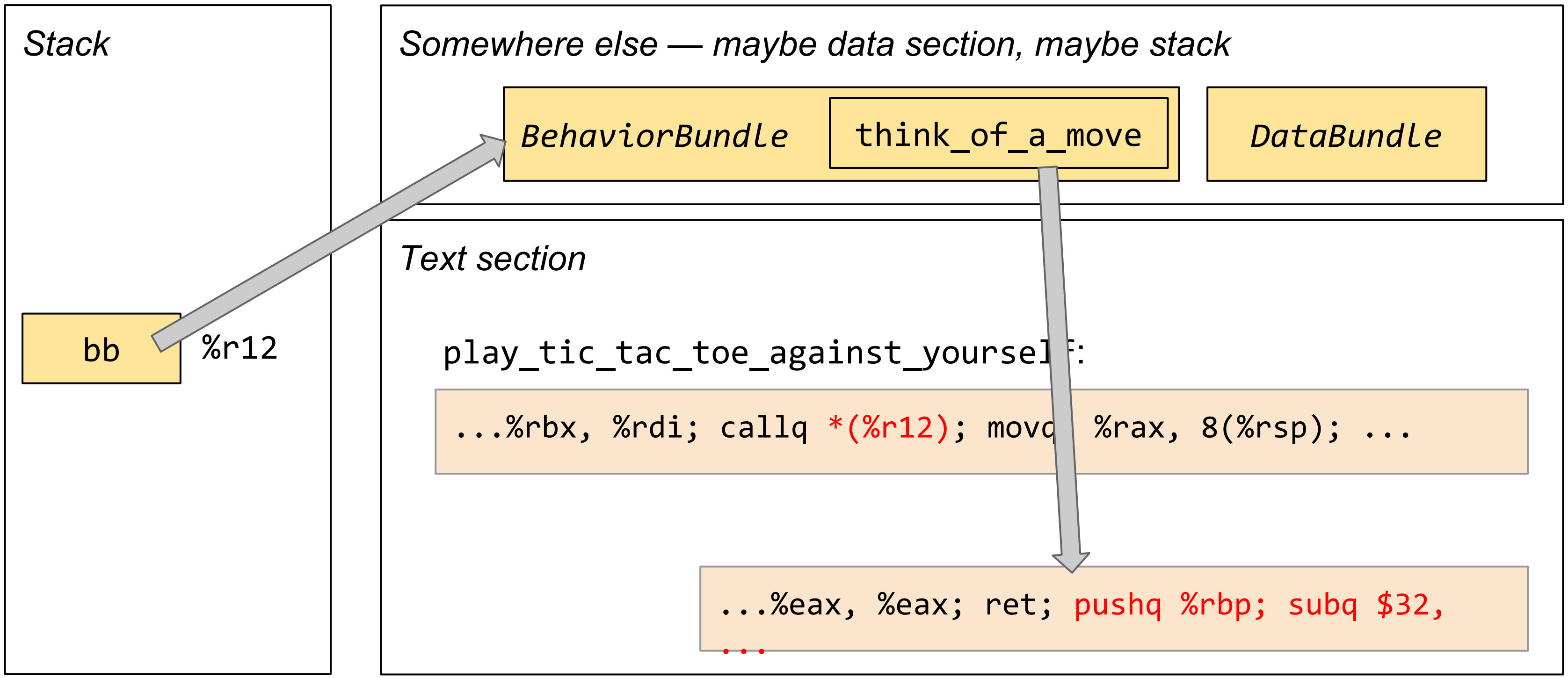


- What if the callback function `think_of_a_move` needs some extra information? That is, what if ***its*** behavior is parametrized by some “adverb”?
- We could pass in a “cookie” containing that adverbial information. (STL things that work like this: `qsort_r()`.)
- What if we have ***lots*** of related behaviors that might want to change?
- We could make ***lots*** of new function parameters. (Ugh.)

Let’s fix both of those problems.

```
using TOAM_t = Move(const Grid&, DataBundle *cookie);

struct DataBundle { ... };
struct BehaviorBundle { TOAM_t *think_of_a_move; };

auto play_tic_tac_toe_against_yourself(Grid position,
                                       BehaviorBundle *bb, DataBundle *db)
{
    while (!winner(position)) {
        Move m = bb->think_of_a_move(position, db);
        position += m;
    }
    return winner(position);
}
```

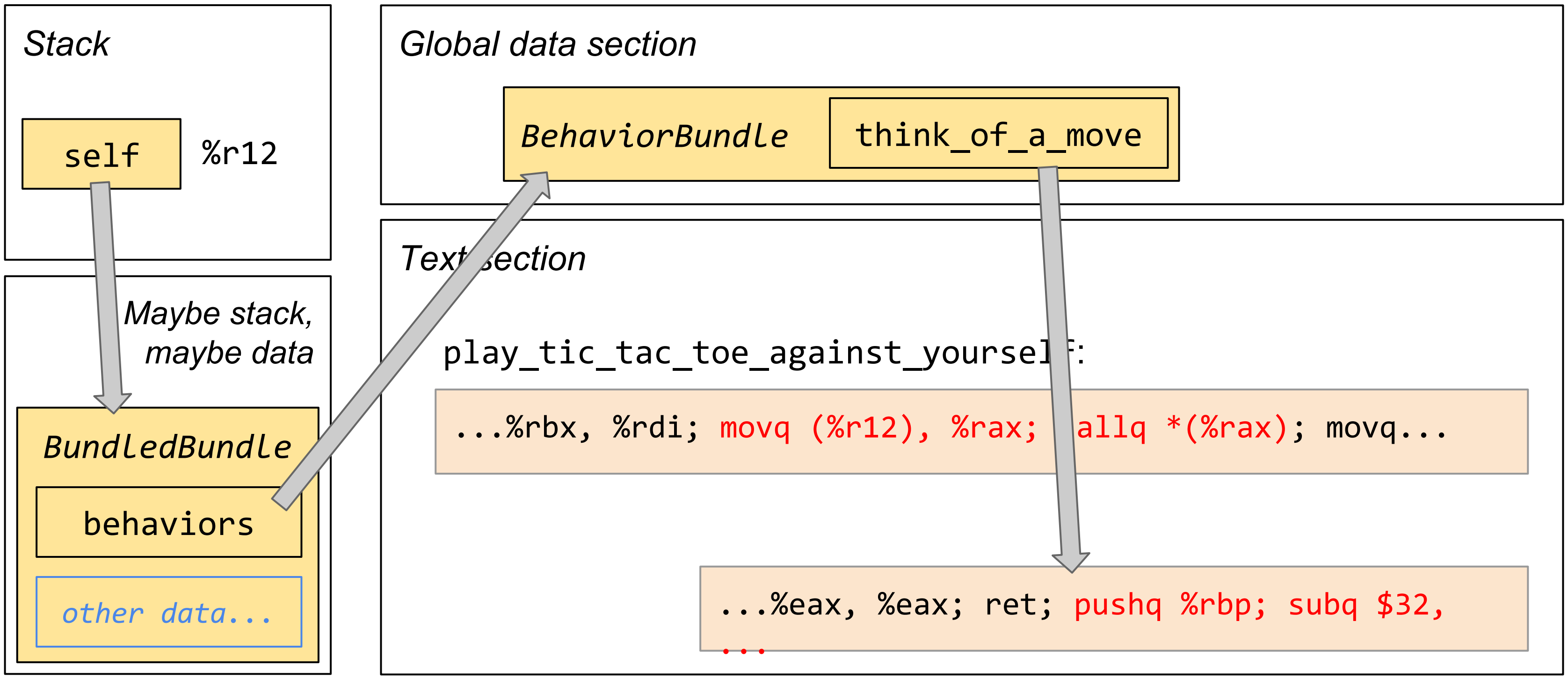
Let’s fix both of those problems.



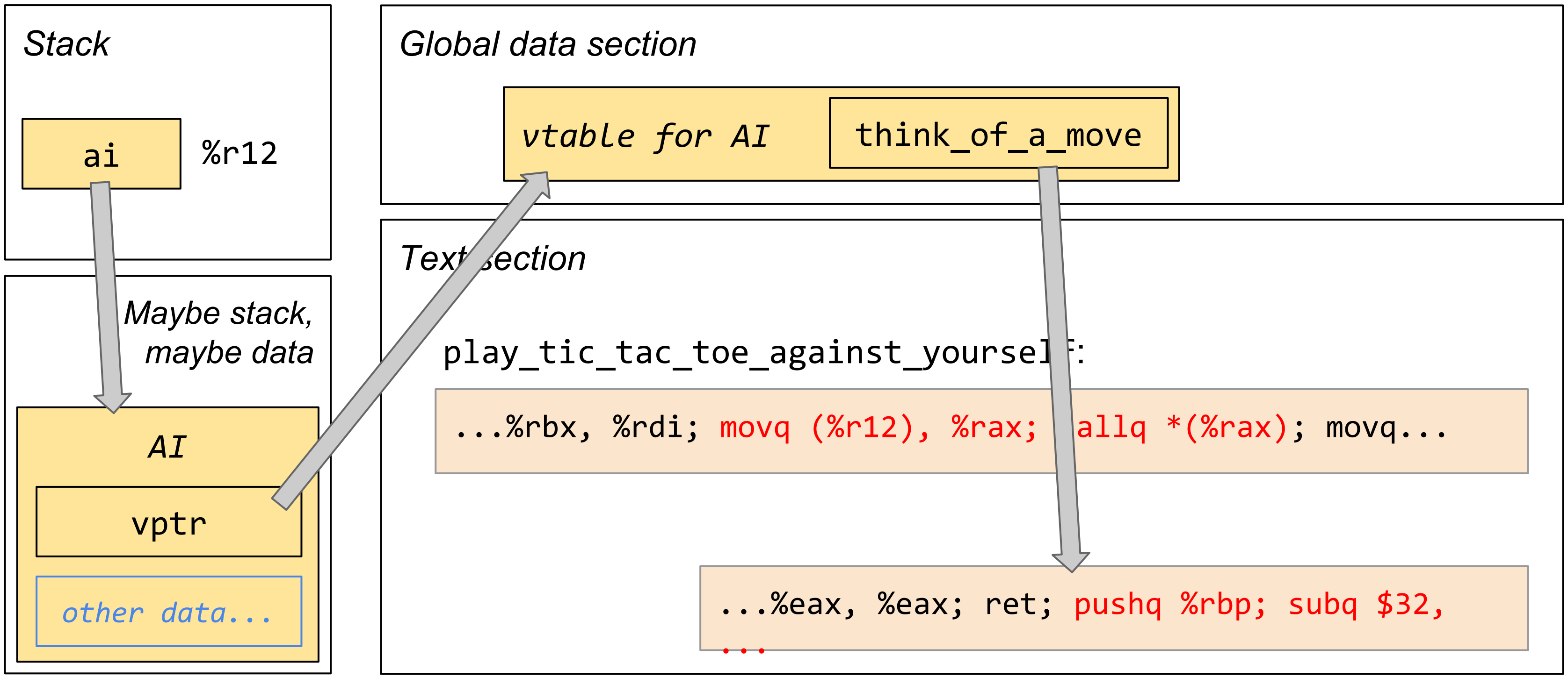
We have invented classical OOP!

```
using TOAM_t = Move(BundledBundle *self, const Grid&);

struct BehaviorBundle { TOAM_t *think_of_a_move; ...other behaviors... };
struct BundledBundle { BehaviorBundle *behaviors; ...other data... };

auto play_tic_tac_toe_against_yourself(BundledBundle *self, Grid position)
{
    while (!winner(position)) {
        Move m = self->behaviors->think_of_a_move(self, position);
        position += m;
    }
    return winner(position);
}
```

We have invented classical OOP!



We have invented classical OOP!



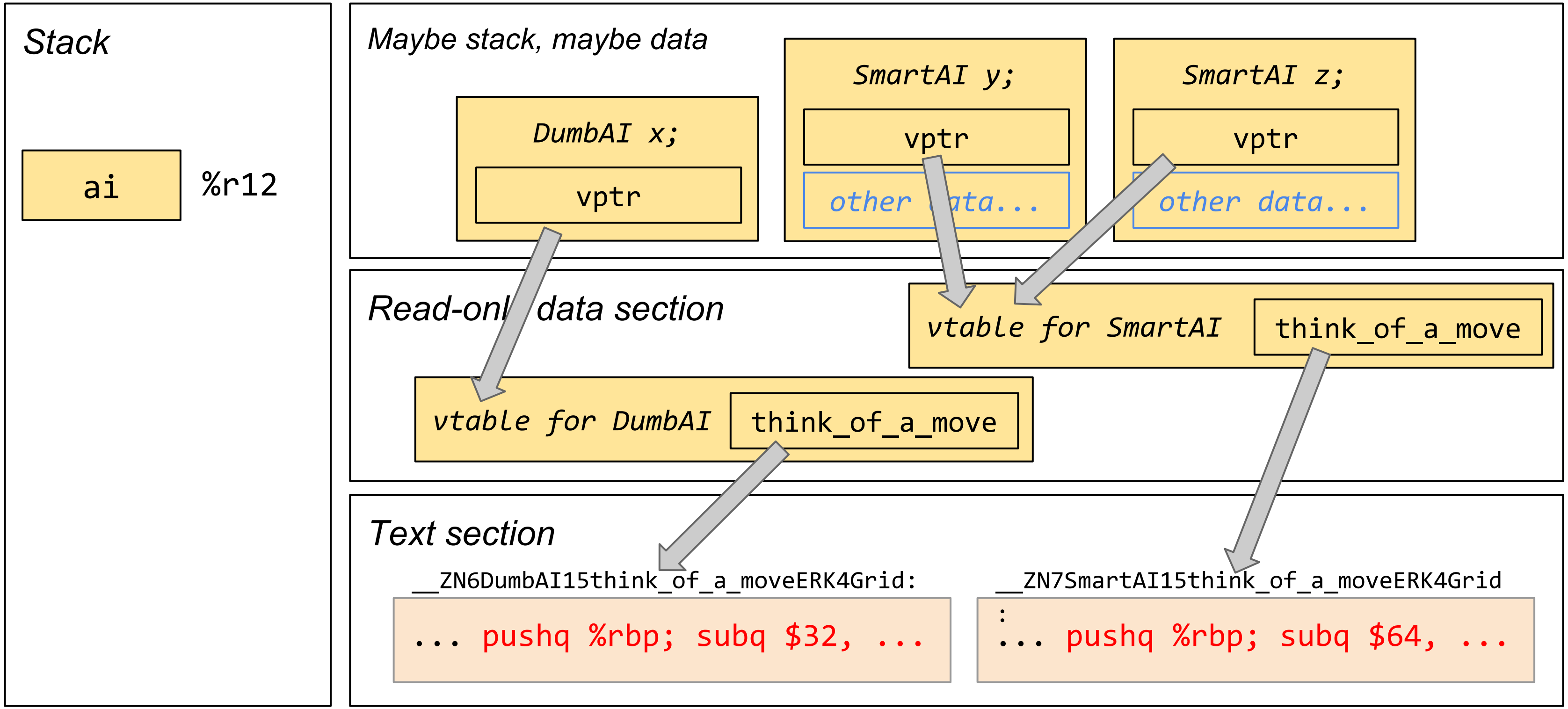
We have invented classical OOP!

```
struct AI {
    virtual Move think_of_a_move(const Grid&) = 0;
    ...other behaviors...
    ...other data...
};

auto play_tic_tac_toe_against_yourself(AI *ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai->think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

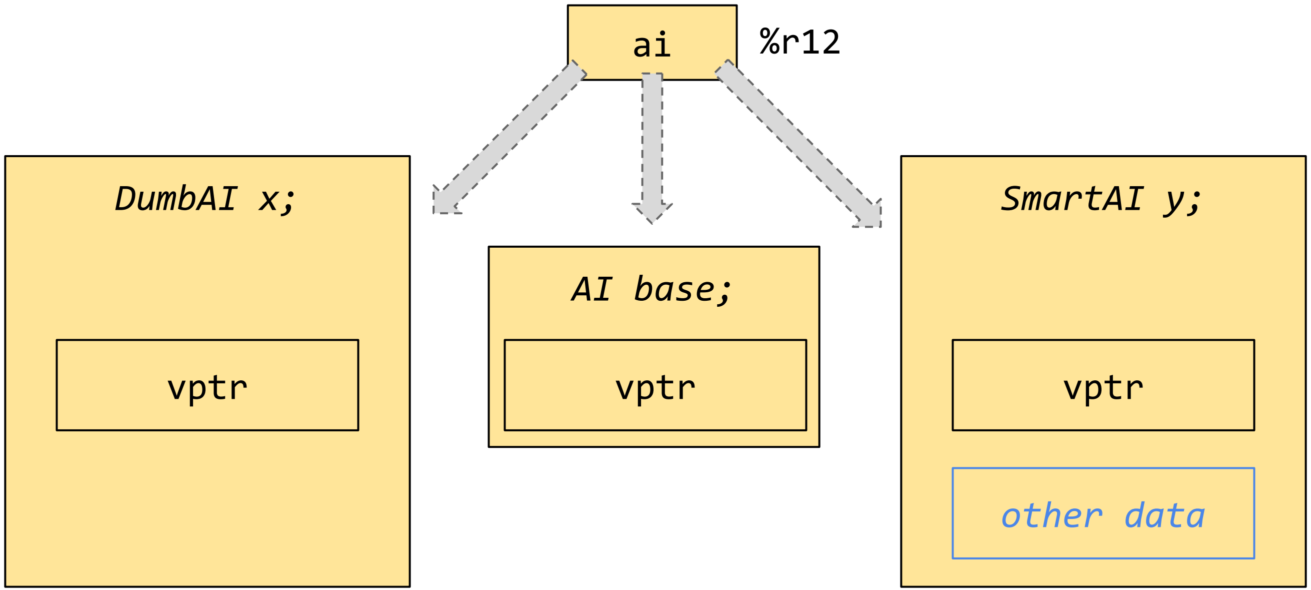
We have invented classical OOP!

```
struct AI {
    virtual Move think_of_a_move(const Grid&) = 0;
    ...other behaviors...
    ...other data...
};

struct DumbAI : public AI {
    Move think_of_a_move(const Grid& g) override { pick a move at random }
};

struct SmartAI : public AI {
    Move think_of_a_move(const Grid& g) override { pick a move cleverly }
};
```

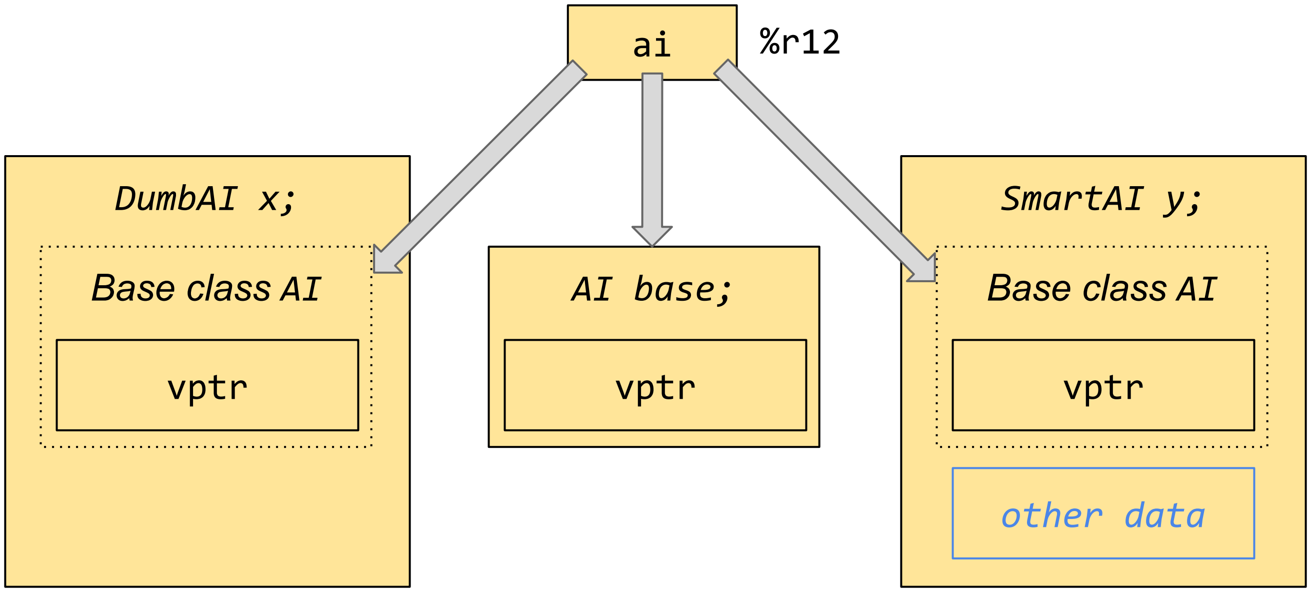
We have invented classical OOP!



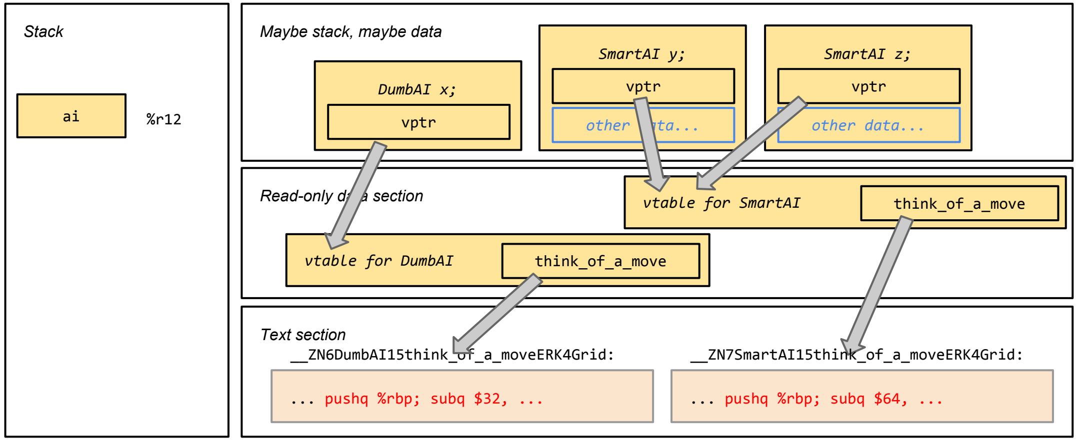
IS-A relationships



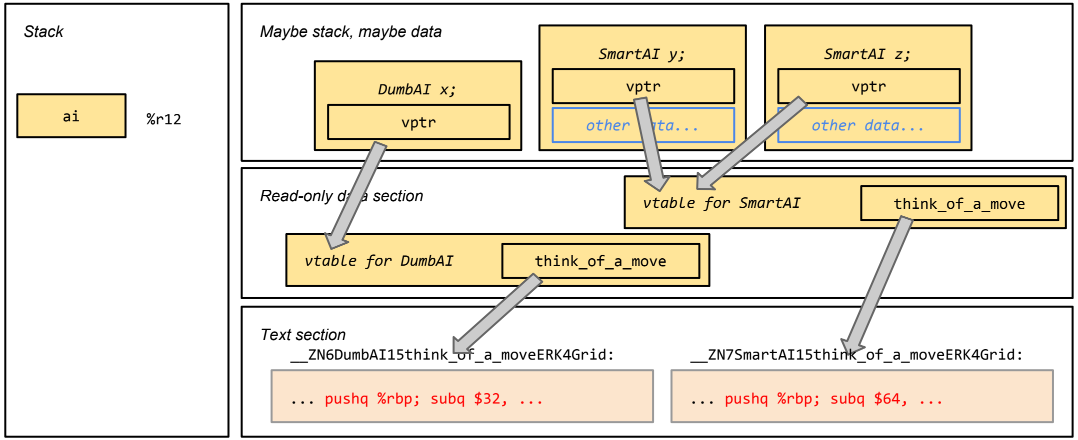
IS-A relationships



Problems with this approach?

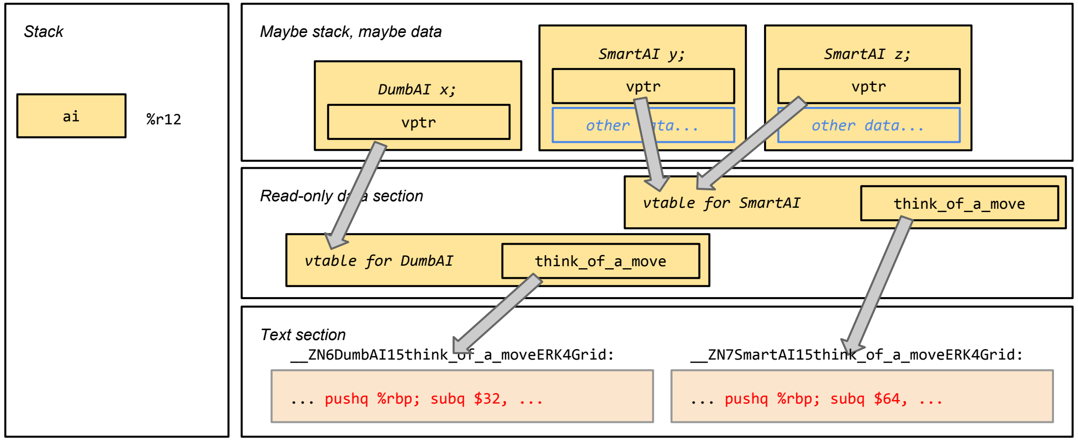


Problems with this approach?



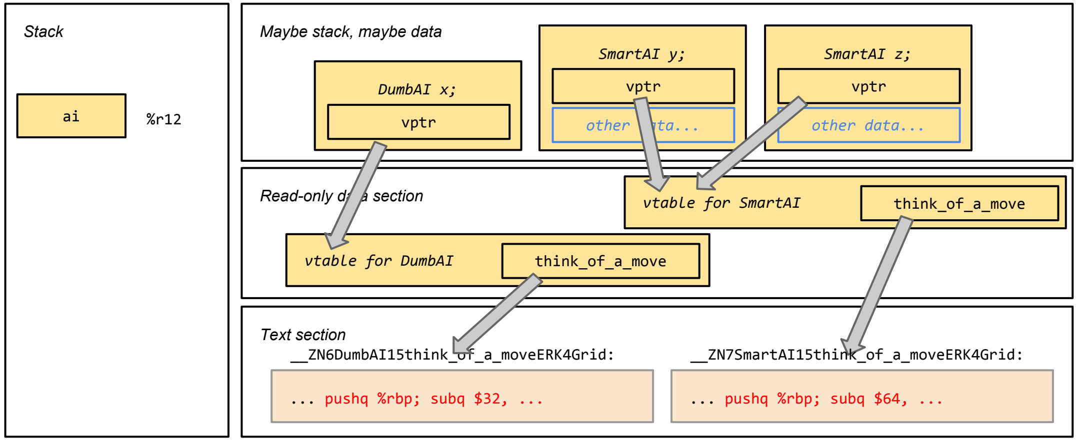
- Pointers everywhere! Pointers are slow, right?

Problems with this approach?



- Pointers everywhere! Pointers are slow, right?
- Worse: a call through a function pointer cannot be inlined, so the optimizer will never get a chance to work on it.

Problems with this approach?



- Pointers everywhere! Pointers are slow, right?
- Worse: a call through a function pointer cannot be inlined, so the optimizer will never get a chance to work on it.
- Semantically, we cannot use our `play_tic_tac_toe` function with arbitrary `think_of_a_move`s anymore, but only with those that have been enrobed in a C++ class that **IS-A** `AI`.

Let’s fix all these problems with templates.

Let’s fix all these problems.

Text section

play_tic_tac_toe_against_yourself:

```
...%rbx, %rdi; callq __Z15think_of_a_moveRK4Grid; movq  %rax, 8(%rsp); ...
```

think_of_a_move:

```
...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; movq %rdi, ...
```

Here the address of `think_of_a_move` is 100% hard-coded into the `play_tic_tac_toe_against_yourself` function.

STL things that work like this: `operator new()`

Let’s fix all these problems.

```
std::optional<Player> winner(const Grid&);
using TOAM_t = Move(const Grid&);
Grid& operator+=(Grid&, const Move&);

template<TOAM_t& think_of_a_move>
auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

Let’s fix all these problems.

```
Move dumb_thinker(const Grid& position) { pick a move at random }

template auto play_tic_tac_toe_against_yourself<dumb_thinker>(Grid);
```

Text section

```
__Z33play_tic_tac_toe_against_yourselfIL_Z12dumb_thinkerRK4GridEEDaS0_:
...%rbx, %rdi; callq __Z12dumb_thinkerRK4Grid; movq  %rax, 8(%rsp); ...

dumb_thinker:
...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; movq %rdi,
...
```

Here the address of dumb_thinker is 100% hard-coded into play_tic_tac_toe_against_yourself<dumb_thinker>.

Let’s fix all these problems.

```
template auto play_tic_tac_toe_against_yourself<dumb_thinker>(Grid);
template auto play_tic_tac_toe_against_yourself<smart_thinker>(Grid);
```

Text section

```
__Z33play_tic_tac_toe_against_yourselfIL_Z12dumb_thinkerRK4GridEEDaS0_:
    ...%rbx, %rdi; callq __Z12dumb_thinkerRK4Grid; movq  %rax, 8(%rsp); ...

__Z33play_tic_tac_toe_against_yourselfIL_Z13smart_thinkerRK4GridEEDaS0_:
    ...%rbx, %rdi; callq __Z13smart_thinkerRK4Grid; movq  %rax, 8(%rsp); ...

smart_thinker:
    ...ret; pushq %rbp; subq $64, %rsp; movq %rdi, ...
```

...And the address of `smart_thinker` is 100% hard-coded into play_tic_tac_toe_against_yourself<smart_thinker>.

STL things that work like this: *???*

Problems with this approach?

```
std::optional<Player> winner(const Grid&);
using TOAM_t = Move(const Grid&);
Grid& operator+=(Grid&, const Move&);

template<TOAM_t& think_of_a_move>
auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

Problems with this approach?

```
std::optional<Player> winner(const Grid&);
using TOAM_t = Move(const Grid&);
Grid& operator+=(Grid&, const Move&);

template<TOAM_t& think_of_a_move>
auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- What if we have ***lots*** of related behaviors that might want to change?
- We could give our function template ***lots*** of template non-type parameters. (Ugh.)

Problems with this approach?

```
std::optional<Player> winner(const Grid&);
using TOAM_t = Move(const Grid&);
Grid& operator+=(Grid&, const Move&);

template<TOAM_t& think_of_a_move>
auto play_tic_tac_toe_against_yourself(Grid position)
{
    while (!winner(position)) {
        Move m = think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- What if we have ***lots*** of related behaviors that might want to change?
- We could give our function template ***lots*** of template non-type parameters. (Ugh.)
- Semantically, we cannot use our `play_tic_tac_toe` function with arbitrary `think_of_a_moves`, but only with those that have been enrobed in a function pointer with this exact signature.

We’ve invented generic programming!

```
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}

struct DumbAI {
    static Move think_of_a_move(Grid position) { pick a move at random }
};

struct SmartAI {
    Move think_of_a_move(const Grid& position) const { pick a move cleverly }
};
```

Let's look at the assembly code.

Text section

```
__Z33play_tic_tac_toe_against_yourselfI6DumbAIEDaRKT_4Grid:
...%rdi; callq __ZN6DumbAI15think_of_a_moveE4Grid; movq %rax, 8(%rsp); ...
...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; ...

__Z33play_tic_tac_toe_against_yourselfI7SmartAIEDaRKT_4Grid:
...%rbx, %rsi; callq __ZNK7SmartAI15think_of_a_moveERK4Grid; movq %rax...
...ret; pushq %rbp; subq $64, %rsp; movq %rdi, ...
```

Super inliner friendly! STL things that work like this: *basically the entire STL*

Problems with this approach?

Text section

```
__Z33play_tic_tac_toe_against_yourselfI6DumbAIEDaRKT_4Grid:

...%rdi; callq __ZN6DumbAI15think_of_a_moveE4Grid; movq %rax, 8(%rsp); ...

...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; ...

__Z33play_tic_tac_toe_against_yourselfI7SmartAIEDaRKT_4Grid:

...%rbx, %rsi; callq __ZNK7SmartAI15think_of_a_moveERK4Grid; movq %rax...

...ret; pushq %rbp; subq $64, %rsp; movq %rdi, ...
```

```
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

Problems with this approach?

Text section

```
__Z33play_tic_tac_toe_against_yourselfI6DumbAIEDaRKT_4Grid:

...%rdi; callq __ZN6DumbAI15think_of_a_moveE4Grid; movq %rax, 8(%rsp); ...

        ...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; ...

__Z33play_tic_tac_toe_against_yourselfI7SmartAIEDaRKT_4Grid:

...%rbx, %rsi; callq __ZNK7SmartAI15think_of_a_moveERK4Grid; movq %rax...

        ...ret; pushq %rbp; subq $64, %rsp; movq %rdi, ...
```

```
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- Lots of repetition at the machine code level: “code bloat.”

Problems with this approach?

Text section

```
__Z33play_tic_tac_toe_against_yourselfI6DumbAIEDaRKT_4Grid:
    ...%rdi; callq __ZN6DumbAI15think_of_a_moveE4Grid; movq %rax, 8(%rsp); ...

    ...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; ...

__Z33play_tic_tac_toe_against_yourselfI7SmartAIEDaRKT_4Grid:
    ...%rbx, %rsi; callq __ZNK7SmartAI15think_of_a_moveERK4Grid; movq %rax...

    ...ret; pushq %rbp; subq $64, %rsp; movq %rdi, ...
```

```
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- Lots of repetition at the machine code level: “code bloat.”
- “Calling” `play_tic_tac_toe` requires instantiating a whole new machine-code version of it. So the implementation must be in a header file. Can’t ship it in a binary ABI.

Problems with this approach?

Text section

```
__Z33play_tic_tac_toe_against_yourselfI6DumbAIEDaRKT_4Grid:
...%rdi; callq __ZN6DumbAI15think_of_a_moveE4Grid; movq %rax, 8(%rsp); ...
...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; ...
__Z33play_tic_tac_toe_against_yourselfI7SmartAIEDaRKT_4Grid:
...%rbx, %rsi; callq __ZNK7SmartAI15think_of_a_moveERK4Grid; movq %rax...
...ret; pushq %rbp; subq $64, %rsp; movq %rdi, ...
```

```
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- Lots of repetition at the machine code level: “code bloat.”
- “Calling” `play_tic_tac_toe` requires instantiating a whole new machine-code version of it. So the implementation must be in a header file. Can’t ship it in a binary ABI.
- Semantically, our `play_tic_tac_toe` function template still has *some* kind of implicit “contract” with its `think_of_a_move` subroutine, but there’s no explicit structure in our C++ code that reifies or documents this contract.

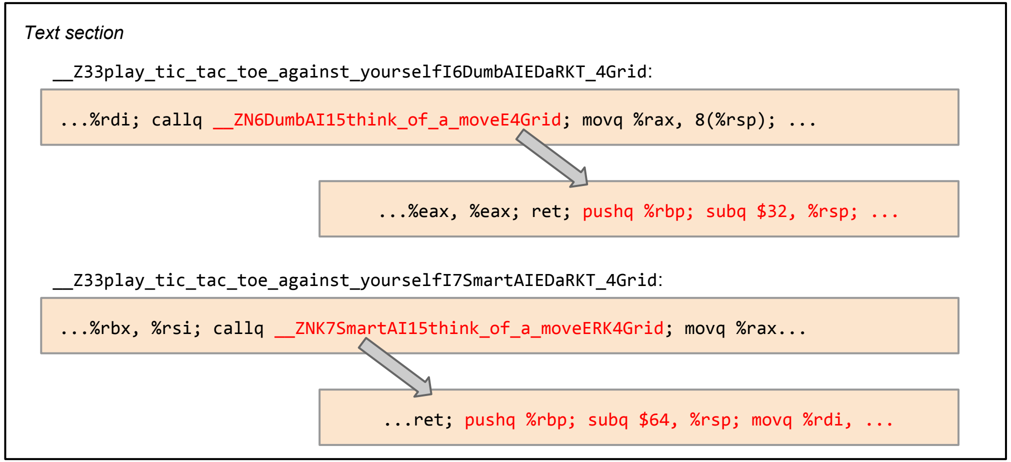
Advantages of this approach?

Text section

```
__Z33play_tic_tac_toe_against_yourselfI6DumbAIEDaRKT_4Grid:
...%rdi; callq __ZN6DumbAI15think_of_a_moveE4Grid; movq %rax, 8(%rsp); ...
...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; ...

__Z33play_tic_tac_toe_against_yourselfI7SmartAIEDaRKT_4Grid:
...%rbx, %rsi; callq __ZNK7SmartAI15think_of_a_moveERK4Grid; movq %rax...
...ret; pushq %rbp; subq $64, %rsp; movq %rdi, ...
```

```
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

Advantages of this approach?




```
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- Efficiency: the code is repeated with only minor variations, but the optimizer gets to run on each repetition individually.

Advantages of this approach?

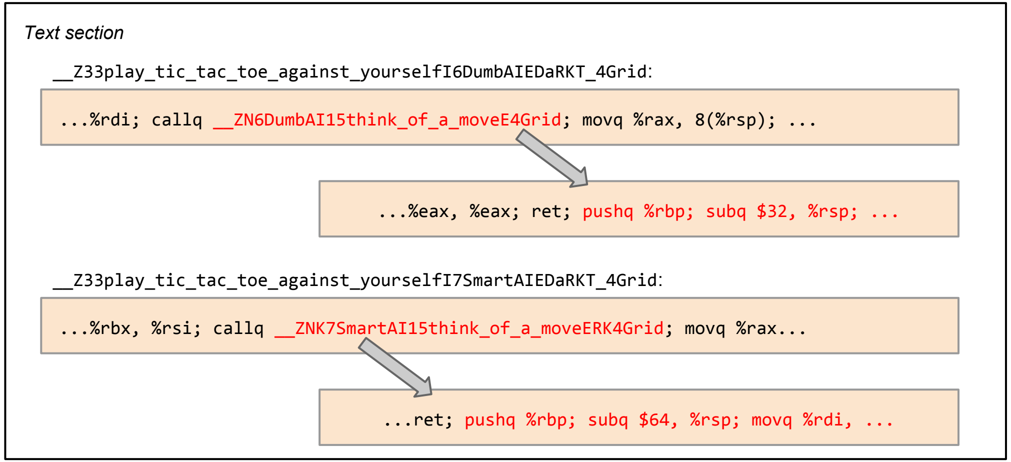
Text section

```
__Z33play_tic_tac_toe_against_yourselfI6DumbAIEDaRKT_4Grid:
...%rdi; callq __ZN6DumbAI15think_of_a_moveE4Grid; movq %rax, 8(%rsp); ...
                    ...%eax, %eax; ret; pushq %rbp; subq $32, %rsp; ...

__Z33play_tic_tac_toe_against_yourselfI7SmartAIEDaRKT_4Grid:
...%rbx, %rsi; callq __ZNK7SmartAI15think_of_a_moveERK4Grid; movq %rax...
                    ...ret; pushq %rbp; subq $64, %rsp; movq %rdi, ...
```

```cpp
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- Efficiency: the code is repeated with only minor variations, but the optimizer gets to run on each repetition individually.
- Semantically, if the contract between `play_tic_tac_toe` and its `think_of_a_move` subroutine is very *complicated*, maybe it's a *good* thing that we don't have to reify that contract in code!

Advantages of this approach?




```
template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

- Efficiency: the code is repeated with only minor variations, but the optimizer gets to run on each repetition individually.
- Semantically, if the contract between `play_tic_tac_toe` and its `think_of_a_move` subroutine is very *complicated*, maybe it's a *good* thing that we don't have to reify that contract in code!
- But let's take just a minute to show how we *might* reify such a contract...

Sidenote: Concepts Lite in C++20?

```
template<typename T>
concept bool Thinker = requires(T thinker, Grid g) {
    { thinker.think_of_a_move(g) } -> Move;
};

template<typename AI>
auto play_tic_tac_toe_against_yourself(const AI& ai, Grid position)
    requires( Thinker<AI> )
{
    while (!winner(position)) {
        Move m = ai.think_of_a_move(position);
        position += m;
    }
    return winner(position);
}
```

PARENTAL ADVISORY EXPLICIT CONTENT

This ends up looking an awful lot like a classical-OOP base class, but it's still fundamentally implemented in terms of templates. Notice that `SmartAI::think_of_a_move` and `DumbAI::think_of_a_move` still have different signatures: `static` vs. not, pass-by-const-ref vs. pass-by-value.

bit.ly/concepts419

Reifying a really complex concept

```
template<typename Container>
int generic_count(Container& container) {
    int x = 0;
    for (auto&& item : container) x += 1;
    return x;
}

int polymorphic_count(ContainerBase& container) {
    int x = 0;
    for (auto&& item : container) x += 1;
    return x;
}
```

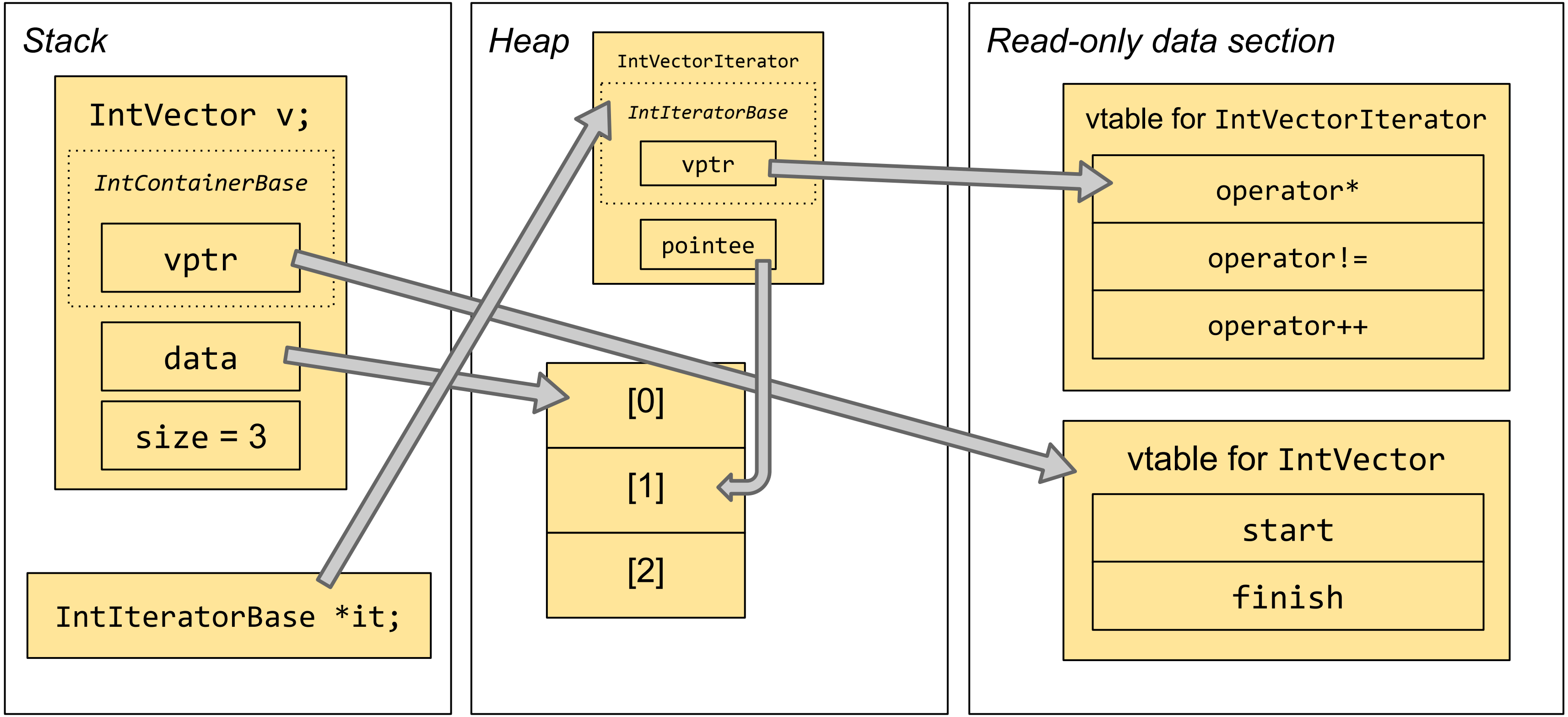
Reifying a really complex concept

```
struct ContainerBase {
    virtual ??? begin() = 0;
    virtual ??? end() = 0;
};

int polymorphic_count(ContainerBase& container) {
    int x = 0;
    for (auto&& item : container) x += 1;
    return x;
}
```

Reifying a really complex concept

```
struct IntContainerBase {
    virtual IntIteratorBase *start() = 0;
    virtual IntIteratorBase *finish() = 0;
};

struct IntIteratorBase {
    virtual int operator* () const = 0;
    virtual bool operator!= (const IntIteratorBase&) const = 0;
    virtual void operator++ () = 0;
};

int polymorphic_count(IntContainerBase& c) {
    int x = 0;
    for (auto it = c.start(); *it != *c.finish(); ++*it) {
        x += 1;
    }
    return x;
}
```

Classical OO all the way



OO doesn't really do "higher-order"

If we continue down this road of *classically polymorphic* algorithms and containers, we soon find we need a "root `Object`" class, which spreads virally.

```
struct IteratorBase {
    Object* operator*() = 0;
};

struct PredicateBase {
    bool operator()(Object* argument) = 0;
};

struct BinaryFunctionBase {
    Object* operator() (Object* a, Object* b) = 0;
};
```

Also notice the complete lack of *value semantics* here. It's all pointers!

So where *do* we see classical OO?

Generally, in places where the interface is *rigidly defined* in a non-generic way. The *behavior* of different derived classes may vary, but we don’t parameterize anything at the *type-system* level.

```
class AI {
    virtual Move think_of_a_move(const Grid&) = 0;
};

class ExpressionNode {
    virtual int evaluate() = 0;
};
```

So where *do* we see classical OO?

Also notice that our `AI` is likely a singleton, and `ExpressionNode` is likely a dynamically allocated tree, which means it's going to be chasing pointers a lot of the time anyway. Neither of these looks like something we'd want to *copy*.

```
class AI {
    virtual Move think_of_a_move(const Grid&) = 0;
};

class ExpressionNode {
    virtual int evaluate() = 0;
};
```

So where *do* we see classical OO?

We also want to use classical OO for calls that must traverse a stable binary ABI boundary. Can't use templates for those! `AI` might be an example, if we want to get our `think_of_a_move` from a shared object file (*`SomeAI.dylib`*). That's also a special case of *switching among different behaviors at runtime*.

```
class AI {
    virtual Move think_of_a_move(const Grid&) = 0;
};

class ExpressionNode {
    virtual int evaluate() = 0;
};
```

STL things that work like this: `std::pmr::memory_resource`

Perfect example: Memory allocation

```
// C++03 STL: “Allocator” is a very complicated concept
template<typename T, typename A>
concept bool Allocator = requires(A alloc, T *ptr, int n) {
    { alloc.allocate(n) } -> T*;
    { alloc.deallocate(ptr, n) } -> void;
    // ... plus many more complicated requirements ...
};

template<class T, class A = std::allocator<T>>
    requires( Allocator<T,A> )
class vector {
{
    // ...
};
```

Case in point: Memory allocation

```
template<class T, class A = std::allocator<T>>
class vector {
{
    // ...
};
```

A big program might have several different strategies for allocating memory. In C++03, this means you write several different classes, each of which models the `Allocator` concept. Then the compiler generates absolutely *tons* of code:

```
std::vector<int, std::allocator<int>>::at(size_t) const
std::vector<int, my::incremental_allocator<int>>::at(size_t) const
std::vector<int, my::slab_allocator<int, 1024>>::at(size_t) const
std::vector<int, my::slab_allocator<int, 4096>>::at(size_t) const
...
```

Case in point: Memory allocation

In the C++17 library, namespace `std::pmr` holds a standard solution for this problem.

```
struct memory_resource {
    void *allocate(size_t bytes, size_t align = MAX_ALIGN) {
        return do_allocate(bytes, align);
    }
    void deallocate(void *p, size_t bytes, size_t align = MAX_ALIGN) {
        return do_deallocate(bytes, align);
    }
    virtual ~memory_resource() = default;
private:
    virtual void *do_allocate(size_t bytes, size_t align) = 0;
    virtual void do_deallocate(void *p, size_t bytes, size_t align) = 0;
};
```

Case in point: Memory allocation

In the C++17 library, namespace `std::pmr` holds a standard solution for this problem.

```
class new_delete_buffer_resource : public memory_resource {
    void *do_allocate(size_t b, size_t) override
        { return new char[b]; }
    void do_deallocate(void * p, size_t, size_t) override
        { delete [] p; }
};

class monotonic_buffer_resource : public memory_resource {
    void *do_allocate(size_t, size_t) override { bump a pointer }
    void do_deallocate(void *, size_t, size_t) override { no-op }
public:
    ~monotonic_buffer_resource() override { free everything }
};
```

Case in point: Memory allocation

Now, if we want to make a function whose memory allocation behavior is parametrizable by the caller, we can just take a pointer to a classically polymorphic `std::pmr::memory_resource`!
Sure, the virtual function call is slow, but memory allocation is generally slow; we might not care about the cost of the call. In return, we get vastly less code bloat —

```
std::vector<int, std::allocator<int>>::at(size_t) const
std::vector<int, my::incremental_allocator<int>>::at(size_t) const
std::vector<int, my::slab_allocator<int, 1024>>::at(size_t) const
std::vector<int, my::slab_allocator<int, 4096>>::at(size_t) const
...

std::vector<int, std::pmr::polymorphic_allocator<int>>::at(size_t) const
```

Problem: `memory_resource*` doesn't satisfy the `Allocator` concept!

```
template<typename T, typename A>
concept bool Allocator = requires(A alloc, T *ptr, int n) {
    { alloc.allocate(n) } -> T*;
    { alloc.deallocate(ptr, n) } -> void;
    // ...
};

struct memory_resource {
    void *allocate(size_t bytes, size_t align = MAX_ALIGN);
    void deallocate(void *p, size_t bytes, size_t align = MAX_ALIGN);
    // ...
};
```

We need a way to *wrap* a `memory_resource*` in an `Allocator`-modeling class object.

Fix the problem with a wrapper class

Namespace `std::pmr` holds a standard solution for *this* problem as well.

```
template<class T>
class polymorphic_allocator {
    memory_resource *mr;
public:
    polymorphic_allocator(memory_resource *mr) : mr(mr) {}

    T *allocate(size_t n) {
        return static_cast<T*>(mr->allocate(n * sizeof(T), alignof(T)));
    }
    void deallocate(T *ptr, size_t n) {
        mr->deallocate(ptr, n * sizeof(T), alignof(T));
    }
};
```

Problem: `polymorphic_allocator` can hold `memory_resource*`s but not `Allocator`s!

```
std::pmr::memory_resource *mr = std::pmr::new_delete_resource();
std::pmr::polymorphic_allocator<int> pa = mr;  // OK

my::incremental_allocator<int> ia;
std::pmr::polymorphic_allocator<int> pa = ia;  // fails!
```

We need a way to *wrap* an `Allocator`-modeling class into a child class of `memory_resource` so that we can use it with functions (or constructors) that take a `memory_resource*`.

Fix the problem with a wrapper class

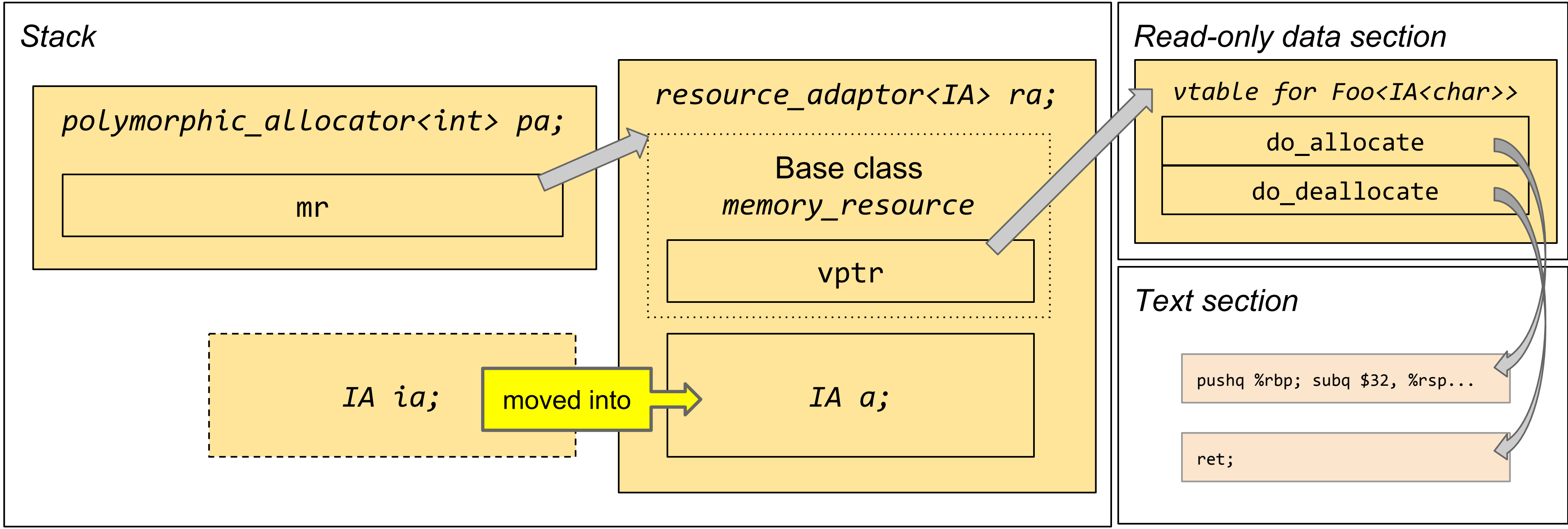
Namespace `std::experimental::pmr` (!) holds a standard solution for this problem.

```
template<class A>
using resource_adaptor = Foo<std::allocator_traits<A>::rebind<char>>;

template<class A>
class Foo : public std::pmr::memory_resource {
    A a;
    void *do_allocate(size_t n, size_t) override {
        return a.allocate(n);
    }
    void do_deallocate(void *p, size_t n, size_t) override {
        return a.deallocate(p, n);
    }
public:
    Foo(A a) : a(std::move(a)) {}
};
```

Other things that work like this: P0260 `queue_wrapper<Q>`

Fix the problem with a wrapper class

```
my::incremental_allocator<int> ia;
std::experimental::pmr::resource_adaptor ra{ std::move(ia) };
std::pmr::polymorphic_allocator<int> pa = &ra;  // OK now!
```

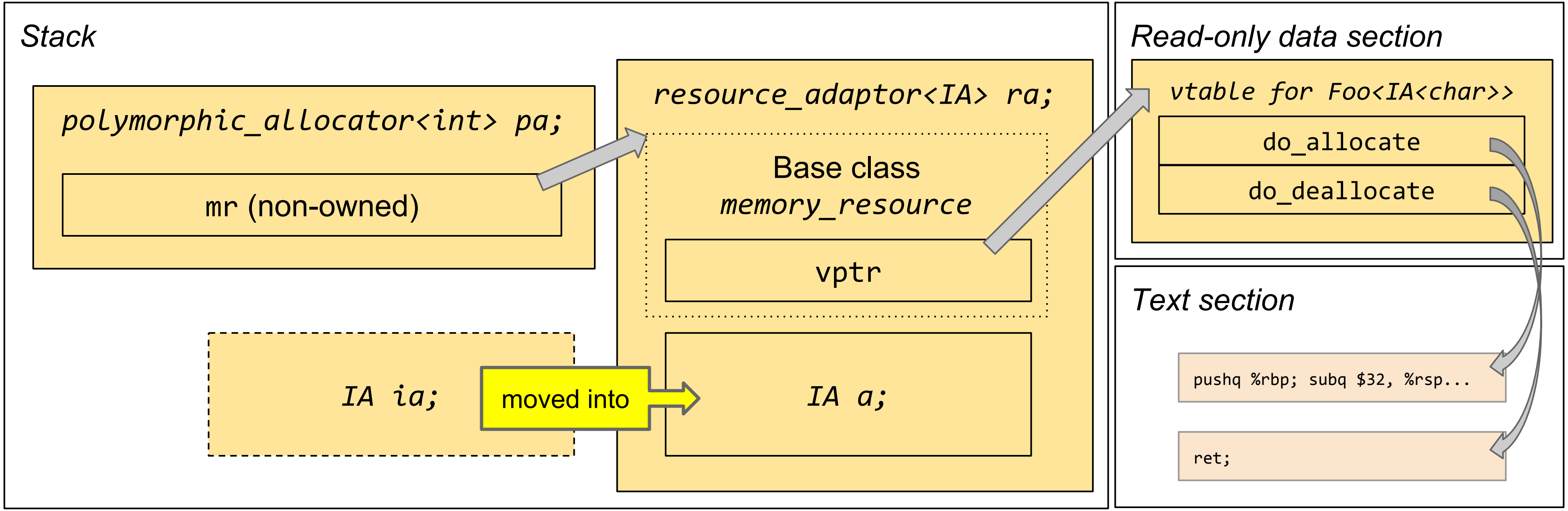


Quick recap of techniques so far

- We've seen how to do classical polymorphism: base class `B`, derived class `D`.
- We've seen how to do generic programming: concept `C`, class `T` *modeling* that concept.
- We've seen how to write a wrapper `Bb` that models `C` but delegates all its behaviors to some `D` at runtime.
- We've seen how to write an adaptor `Db<T>` that inherits from `B` but delegates all its behaviors to `T`.

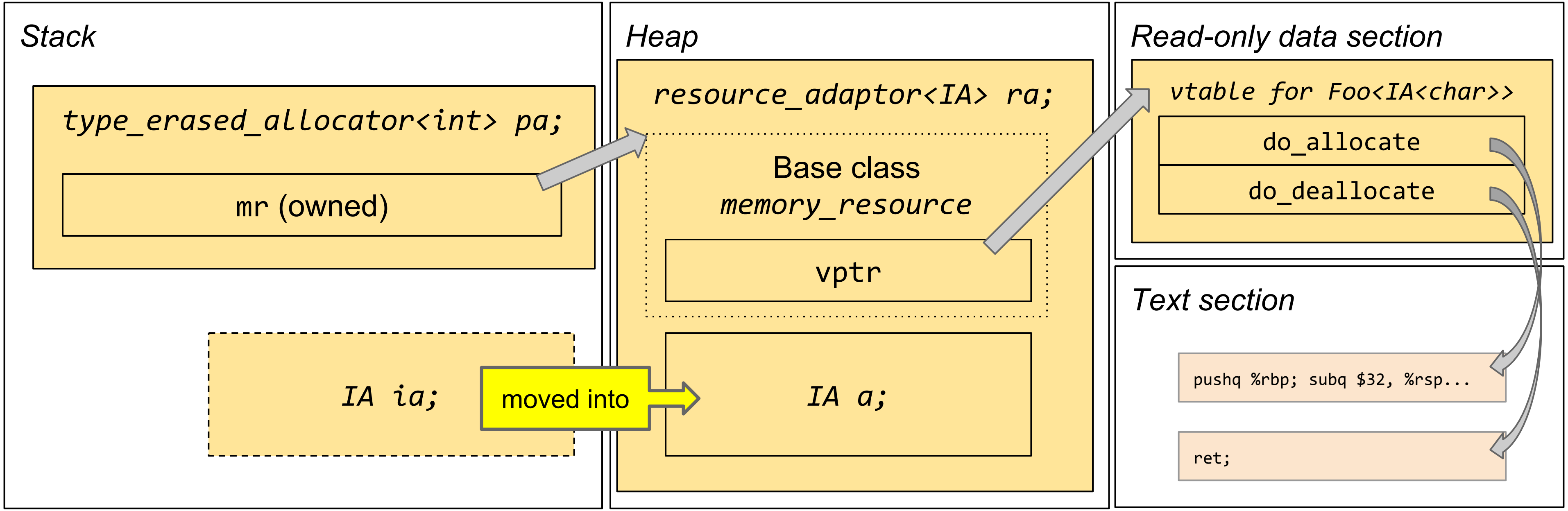
Type erasure is easy now

```
my::incremental_allocator<int> ia;
std::experimental::pmr::resource_adaptor ra{ std::move(ia) };
std::pmr::polymorphic_allocator<int> pa = &ra;  // OK
```



Type erasure is easy now

```
my::incremental_allocator<int> ia;

my::type_erased_allocator<int> pa{ std::move(ia) };
```



STL things that work like this: `std::shared_ptr` deleters, `std::function`, `std::any`

Speeding up virtual dispatch

C++11 introduces the `final` keyword. This means “Nobody inherits from me” (on a class), or “None of my children override this method” (on a method).

If you use `virtual` at all, I recommend you use `final` where possible.

```
struct AI {
    virtual Move think_of_a_move(const Grid&) = 0;
};

struct DumbAI : public AI {
    Move think_of_a_move(const Grid&) override final { pick at random }
};

Move showcase(DumbAI *dai, Grid g) {
    return dai->think_of_a_move(g);  // This virtual call will be inlined!
}
```

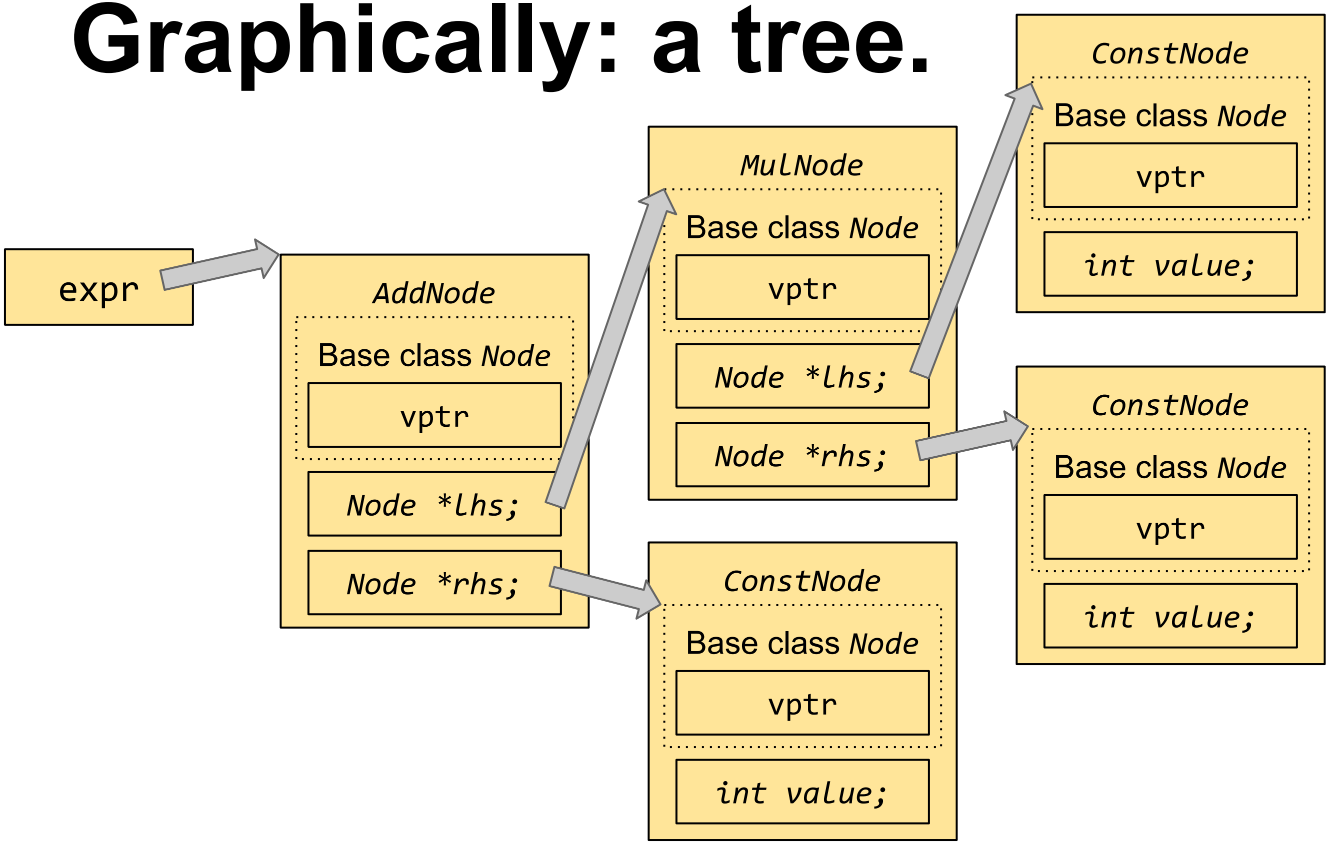
Speeding up virtual dispatch

```
// The following nifty trick with virtuals was shown to me by Louis Dionne.

template<class T> using sptr = std::shared_ptr<T>;

class Node {
public:
    virtual int eval() = 0;
};

class AddNode final : public Node {
    sptr<Node> lhs, rhs;
public:
    AddNode(sptr<Node> l, sptr<Node> r) : lhs(l), rhs(r) {}
    int eval() override { return lhs->eval() + rhs->eval(); }
};
```

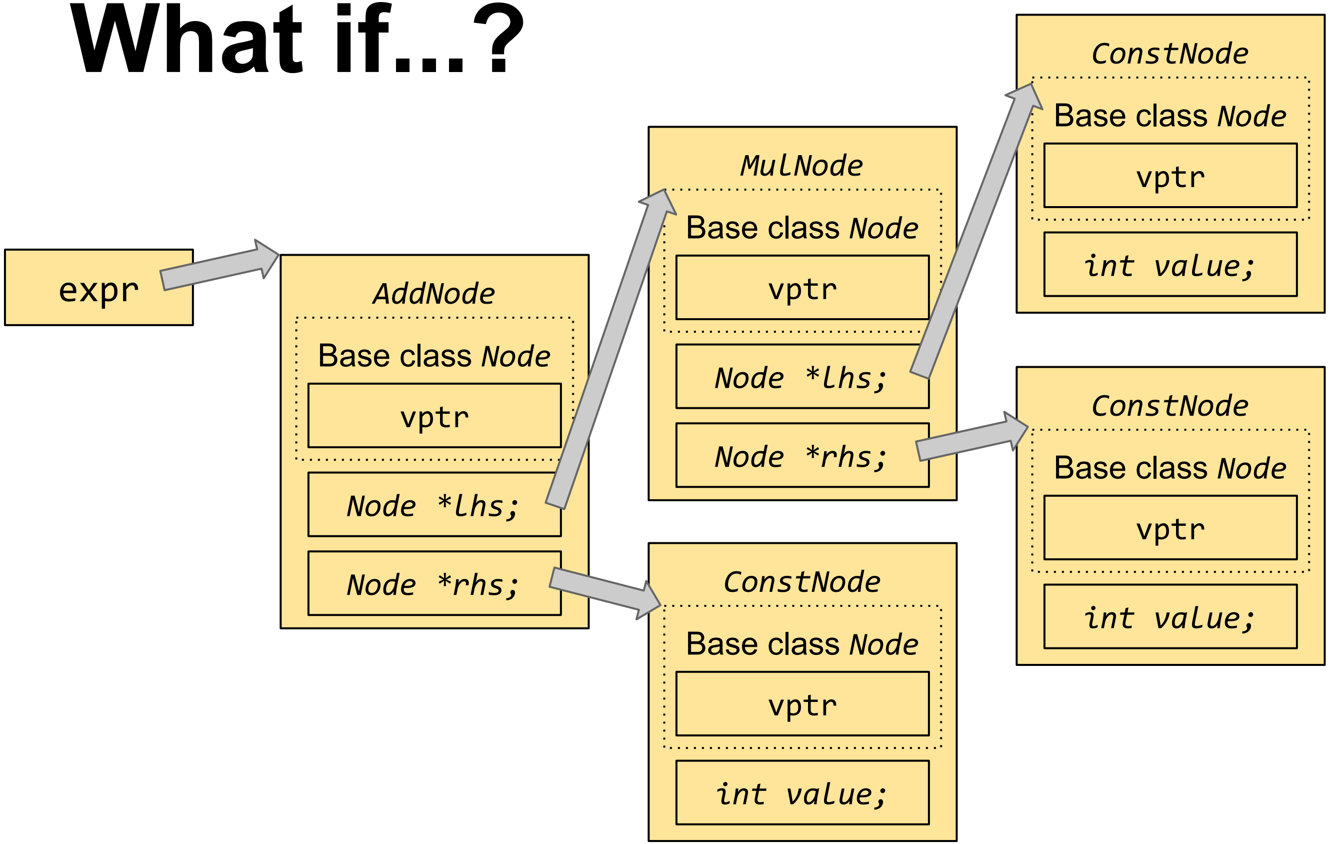
Speeding up virtual dispatch

```
class ConstNode final : public Node {
    int value;
public:
    ConstNode(int v) : value(v) {}
    int eval() override { return value; }
};

int main() {
    auto expr = make_shared<AddNode>(
        make_shared<MulNode>(
            make_shared<ConstNode>(1), make_shared<ConstNode>(2)
        ),
        make_shared<ConstNode>(3)
    );

    int result = expr->eval();
    assert(result == 1*2 + 3);
}
```

Graphically: a tree.

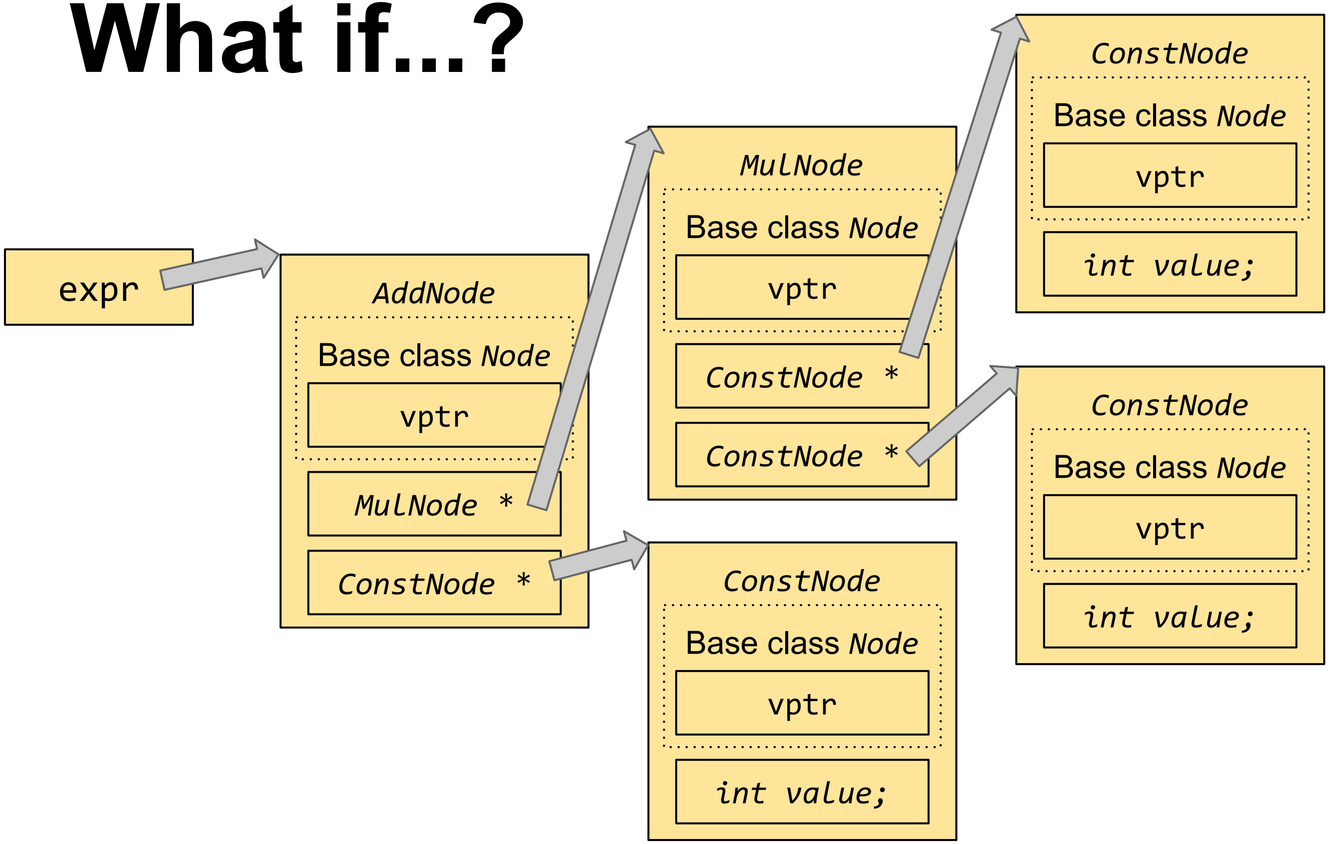


What if...?



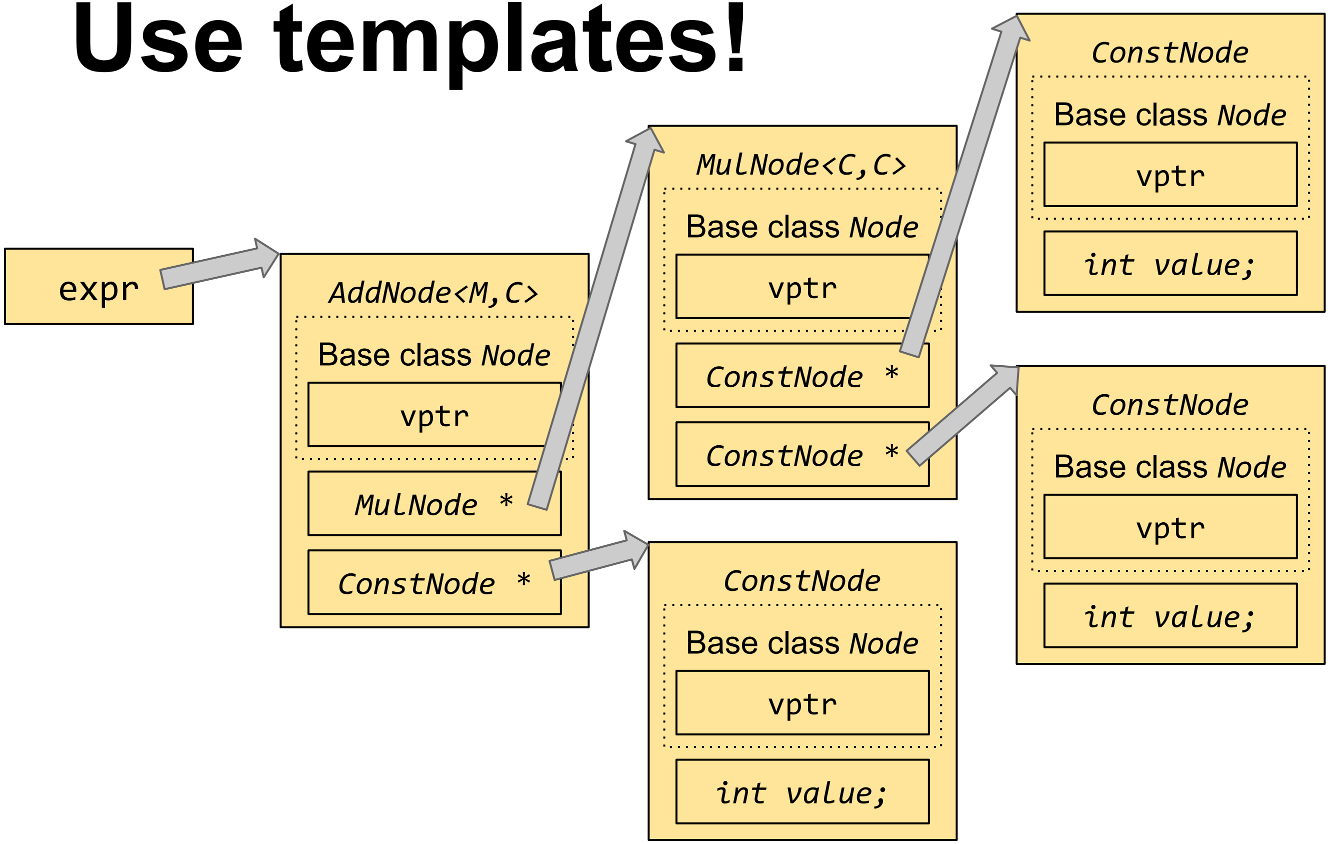
The non-inlineability of each virtual dispatch is due to the fact that each node doesn't know the dynamic type(s) of its `lhs` and `rhs` nodes. What if we gave the nodes that type information somehow?

What if...?



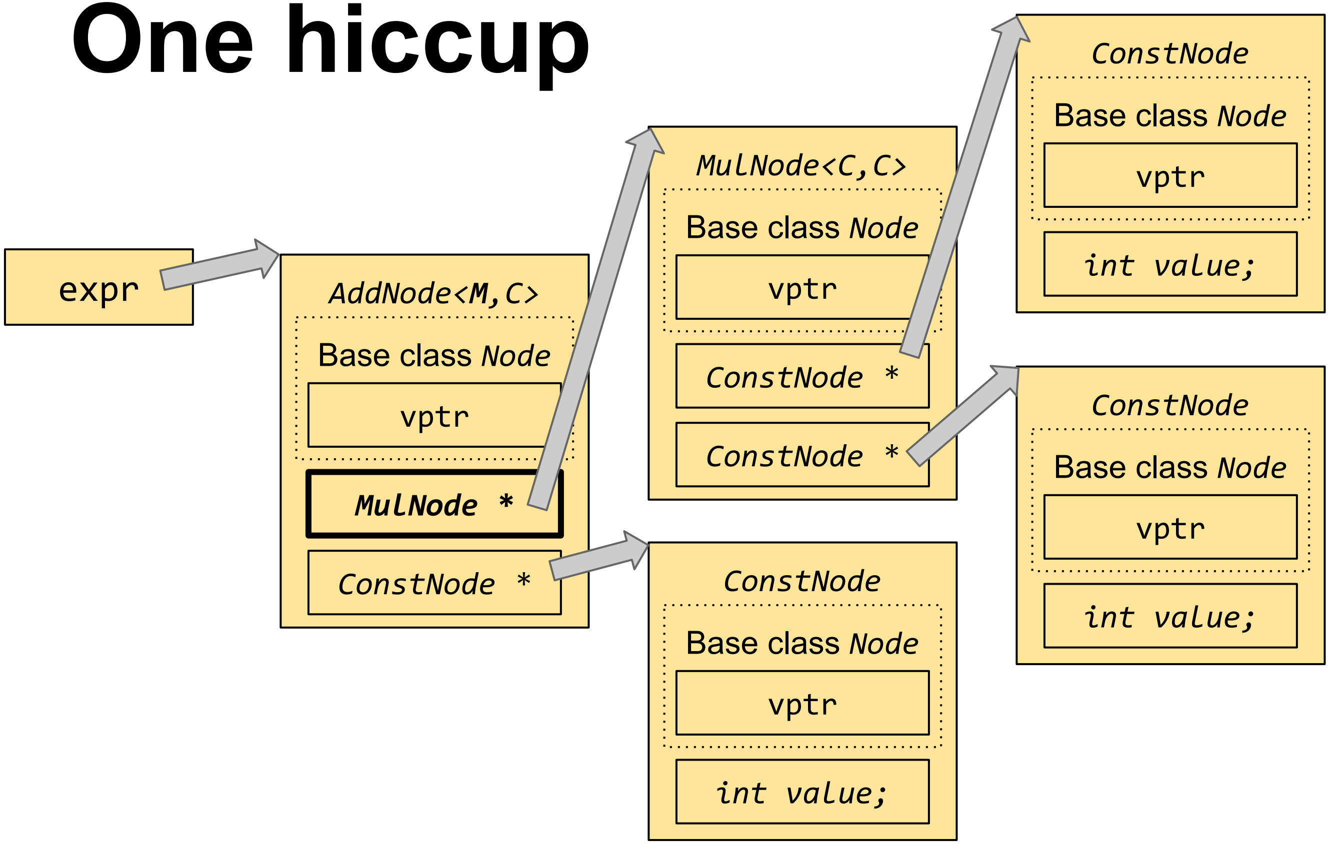
The non-inlineability of each virtual dispatch is due to the fact that each node doesn’t know the dynamic type(s) of its `lhs` and `rhs` nodes. What if we gave the nodes that type information somehow?

Use templates!



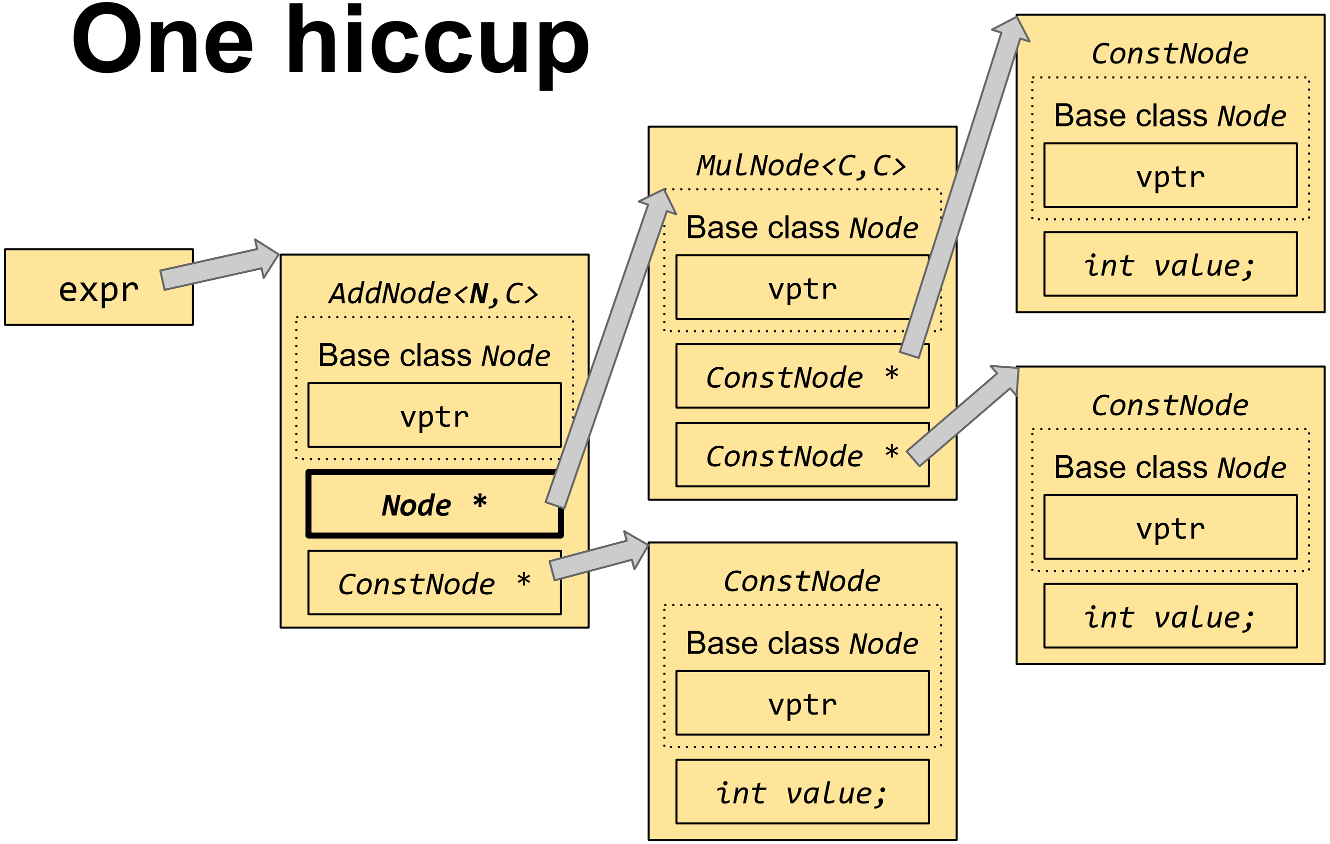
Any problem with this?

One hiccup



The highlighted node is a bit awkward, because `MulNode*` is no longer a valid type; what we mean here is `MulNode<C,C>*`. We're not going to encode the entire tree into the typesystem, are we?

One hiccup



Fortunately, an object of type `MulNode<C,C>` **IS-A** `Node`, so we can store a `Node*` here and resign ourselves to non-inlineable virtual dispatch in this one particular case.

Let’s see the C++ code.

```
class Node {
public:
    virtual int eval() = 0;
};

template<class Left = Node, class Right = Node>
class AddNode final : public Node {
    sptr<Left> lhs;
    sptr<Right> rhs;
public:
    AddNode(sptr<Left> l, sptr<Right> r) : lhs(l), rhs(r) {}
    int eval() override { return lhs->eval() + rhs->eval(); }
};
```

Slow...

```
class ConstNode final : public Node {
    int value;
public:
    ConstNode(int v) : value(v) {}
    int eval() override { return value; }
};

int main() {
    auto expr = make_shared<AddNode<>>(
        make_shared<MulNode<>>(
            make_shared<ConstNode>(1), make_shared<ConstNode>(2)
        ),
        make_shared<ConstNode>(3)
    );

    int result = expr->eval();
    assert(result == 1*2 + 3);
}
```

bit.ly/nifty-tree

Faster...

```
class ConstNode final : public Node {
    int value;
public:
    ConstNode(int v) : value(v) {}
    int eval() override { return value; }
};

int main() {
    auto expr = make_shared<AddNode<Node, ConstNode>>(
        make_shared<MulNode<ConstNode, ConstNode>>(
            make_shared<ConstNode>(1), make_shared<ConstNode>(2)
        ),
        make_shared<ConstNode>(3)
    );

    int result = expr->eval();
    assert(result == 1*2 + 3);
}
```

...Fastest

```
class ConstNode final : public Node {
    int value;
public:
    ConstNode(int v) : value(v) {}
    int eval() override { return value; }
};

int main() {
    auto expr = make_shared<AddNode<MulNode<ConstNode, ConstNode>, ConstNode>>(
        make_shared<MulNode<ConstNode, ConstNode>>(
            make_shared<ConstNode>(1), make_shared<ConstNode>(2)
        ),
        make_shared<ConstNode>(3)
    );

    int result = expr->eval();
    assert(result == 1*2 + 3);
}
```

Testing our speedy tree

```
Wall-clock time for 1 billion executions:

Not at all devirtualized: 10941  milliseconds
Somewhat devirtualized:    5017  milliseconds
Fully devirtualized:       3316  milliseconds
```

Recap and Q&A

We’ve seen:

- How to do classical polymorphism: base class `B`, derived class `D`.
- How to do generic programming: concept `C`, class `T` *satisfying* that concept.
- How to write a wrapper `Bb` that satisfies `C` but delegates all its behaviors to some `D` at runtime.
- How to write an adaptor `Db<T>` that inherits from `B` but delegates all its behaviors to `T`.
- How to unify those adaptor patterns with value semantics by means of *type erasure*.
- That `final` should be used where possible.
- A nifty trick for devirtualizing expression trees.