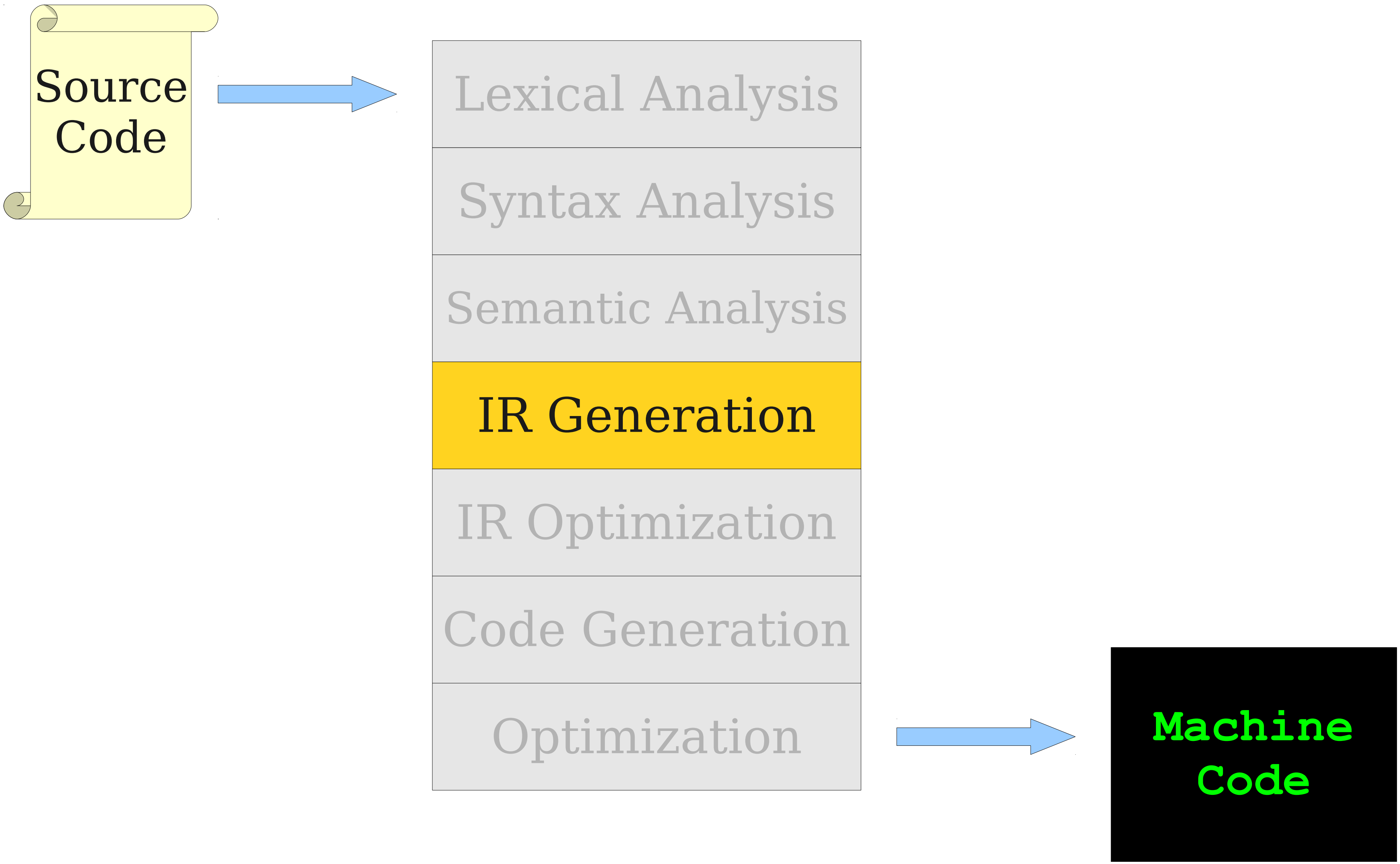
Three-Address Code IR

Announcements

- Programming Project 3 due Monday at 11:59PM.
  - OH today after lecture.
  - Ask questions on Piazzza!
  - Ask questions via email!
- Checkpoint feedback will be returned soon.

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Overview for Today

- The Final Assignment
- Introduction to TAC:
  - TAC for simple expressions.
  - TAC for functions and function calls.
  - TAC for objects.
  - TAC for arrays.
- Generating TAC.
- A few low-level details.

The Final Assignment

- **Goal:** Generate TAC IR for Decaf programs.
- We provide a code generator to produce MIPS assembly.
  - You can run your programs using **`spim`**, the MIPS simulator.
- You must also take care of some low-level details:
  - Assign all parameters, local variables, and temporaries positions in a stack frame.
  - Assign all global variables positions in the global memory segment.
  - Assign all fields in a class an offset from the base of the object.
- You **should not** need to know MIPS to do this; all details will be covered in lecture.
- If you have any questions on MIPS, please feel to ask!

An Important Detail

- When generating IR at this level, you do **not** need to worry about optimizing it.
- It's okay to generate IR that has lots of unnecessary assignments, redundant computations, etc.
- We'll see how to optimize IR code later this week and at the start of next week.
  - It's tricky, but extremely cool!

Three-Address Code

- Or “**TAC**”
- The IR that you will be using for the final programming project.
- High-level assembly where each operation has at most three operands.
- Uses explicit runtime stack for function calls.
- Uses vtables for dynamic dispatch.

Sample TAC Code

```
int x;
int y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;
```

Sample TAC Code

```
int x;
int y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;
```

```
x2 = x * x;
y2 = y * y;
r2 = x2 + y2;
```

Sample TAC Code

```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

Sample TAC Code

```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

```
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```

Sample TAC Code

```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

```
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```

Temporary Variables

- The “three” in “three-address code” refers to the number of operands in any instruction.
- Evaluating an expression with more than three subexpressions requires the introduction of temporary variables.
- This is actually a lot easier than you might think; we'll see how to do it later on.

Sample TAC Code

```
int a;
int b;

a = 5 + 2 * b;
```

Sample TAC Code

```
int a;
int b;

a = 5 + 2 * b;
```

```
_t0 = 5;
_t1 = 2 * b;
a = _t0 + _t1;
```

Sample TAC Code

```
int a;
int b;

a = 5 + 2 * b;
```

TAC allows for instructions with two operands.

```
_t0 = 5;
_t1 = 2 * b;
a = _t0 + _t1;
```

Simple TAC Instructions

- **Variable assignment** allows assignments of the form
  - `var = constant;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$`;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$ **`op`** $\texttt{var}_3$`;`
  - $\texttt{var}_1$ `= constant` **`op`** $\texttt{var}_2$`;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$ **`op`** `constant;`
  - `var =` $\texttt{constant}_1$ **`op`** $\texttt{constant}_2$`;`
- Permitted operators are **`+`**, **`-`**, **`*`**, **`/`**, **`%`**.
- How would you compile **`y = -x`**`;` ?

Simple TAC Instructions

- **Variable assignment** allows assignments of the form
  - `var = constant;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$`;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$ **`op`** $\texttt{var}_3$`;`
  - $\texttt{var}_1$ `= constant` **`op`** $\texttt{var}_2$`;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$ **`op`** `constant;`
  - `var =` $\texttt{constant}_1$ **`op`** $\texttt{constant}_2$`;`
- Permitted operators are **`+`**, **`-`**, **`*`**, **`/`**, **`%`**.
- How would you compile **`y = -x`**`;` ?

`y = 0 - x;` **`y = -1 * x;`**

One More with `bool`s

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

One More with **bools**

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

TAC with `bool`s

- Boolean variables are represented as integers that have zero or nonzero values.
- In addition to the arithmetic operator, TAC supports `<`, `==`, `||`, and `&&`.
- How might you compile `b = (x <= y)` ?

TAC with `bool`s

- Boolean variables are represented as integers that have zero or nonzero values.
- In addition to the arithmetic operator, TAC supports `<`, `==`, `||`, and `&&`.
- How might you compile `b = (x <= y)` ?

```
_t0 = x < y;
_t1 = x == y;
b = _t0 || _t1;
```

Control Flow Statements

```
int x;
int y;
int z;

if (x < y)
   z = x;
else
   z = y;

z = z * z;
```

Control Flow Statements

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```
    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

Control Flow Statements

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```
    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

Control Flow Statements

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```
    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

Labels

- TAC allows for **named labels** indicating particular points in the code that can be jumped to.
- There are two control flow instructions:
  - **`Goto`** ***`label;`***
  - **`IfZ`** ***`value`*** **`Goto`** ***`label;`***
- Note that **`IfZ`** is always paired with **`Goto`**.

Control Flow Statements

```
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;
```

Control Flow Statements

```
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;
```

```
_L0:
    _t0 = x < y;
    IfZ _t0 Goto _L1;
    x = x * 2;
    Goto _L0;
_L1:
    y = x;
```

A Complete Decaf Program

```
void main() {
   int x, y;
   int m2 = x * x + y * y;

   while (m2 > 5) {
      m2 = m2 - x;
   }
}
```

A Complete Decaf Program

```
void main() {
   int x, y;
   int m2 = x * x + y * y;

   while (m2 > 5) {
      m2 = m2 - x;
   }
}
```

```
main:
   BeginFunc 24;
   _t0 = x * x;
   _t1 = y * y;
   m2 = _t0 + _t1;
_L0:
   _t2 = 5 < m2;
   IfZ _t2 Goto _L1;
   m2 = m2 - x;
   Goto _L0;
_L1:
   EndFunc;
```

A Complete Decaf Program

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

```
main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
```

A Complete Decaf Program

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

```
main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
```

A Complete Decaf Program

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

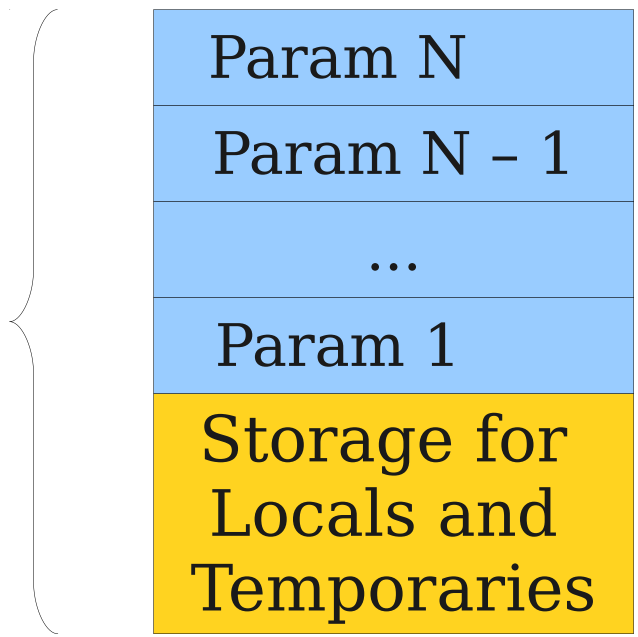
```
main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
```

Compiling Functions

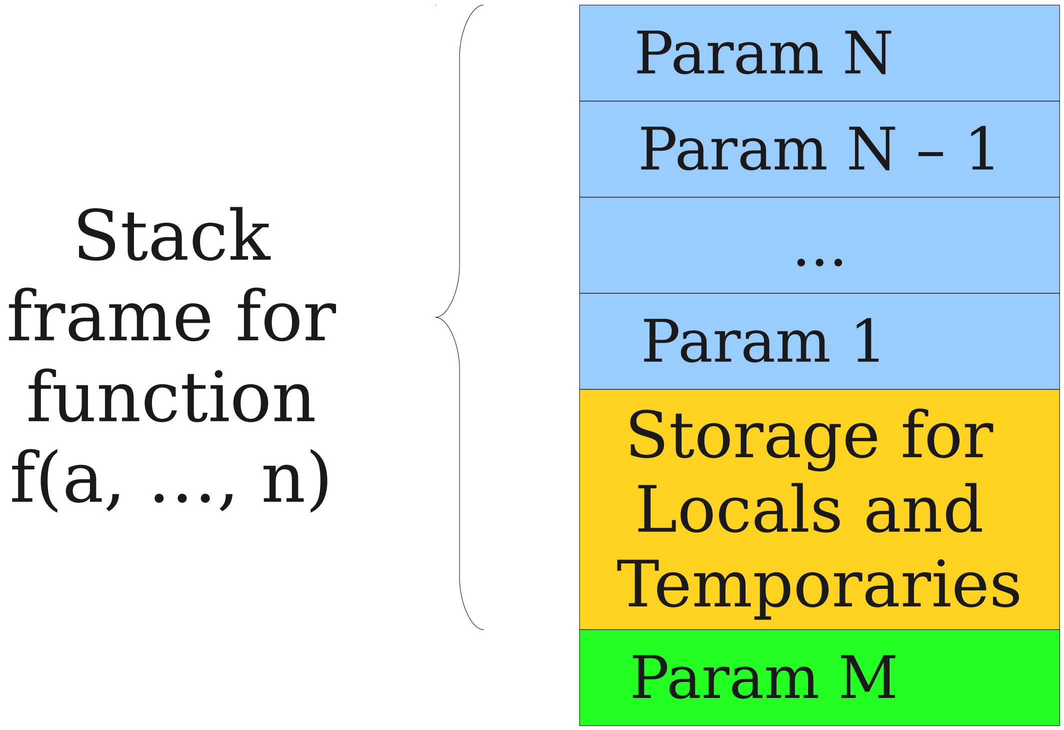
- Decaf functions consist of four pieces:
  - A **label** identifying the start of the function.
    - *(Why?)*
  - A **`BeginFunc N;`** instruction reserving **N** bytes of space for locals and temporaries.
  - The body of the function.
  - An **`EndFunc;`** instruction marking the end of the function.
    - When reached, cleans up stack frame and returns.

A Logical Decaf Stack Frame

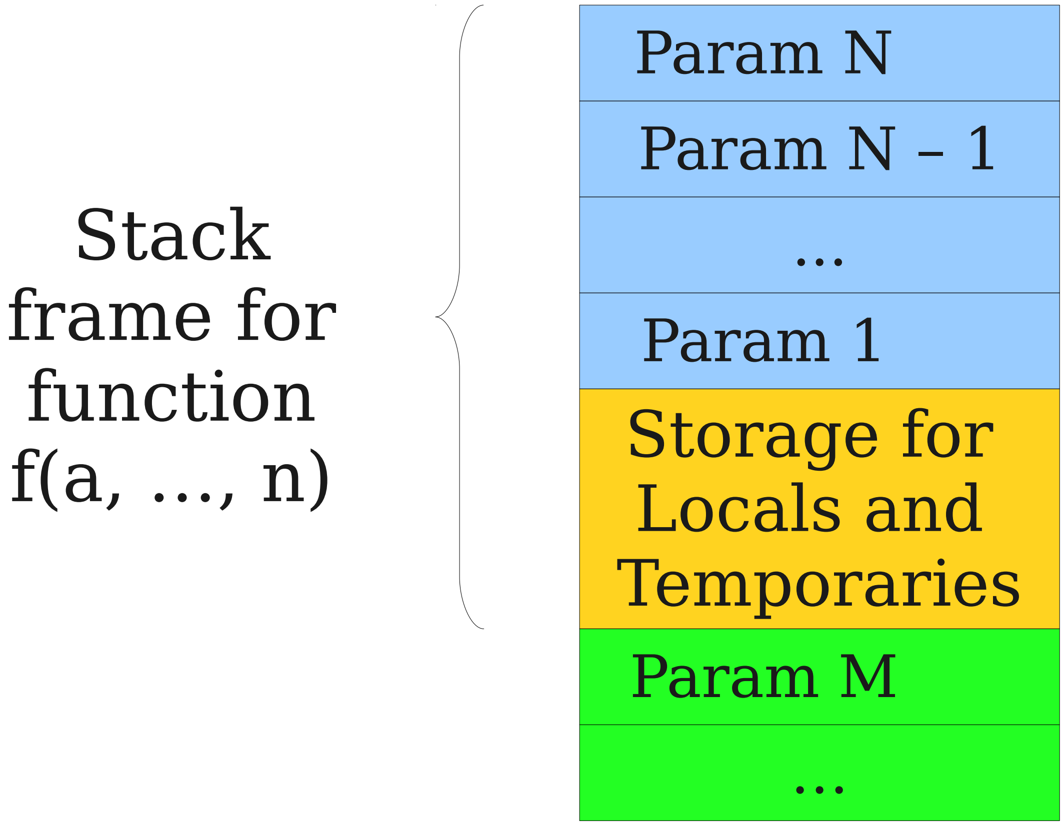
Stack
frame for
function
f(a, …, n)



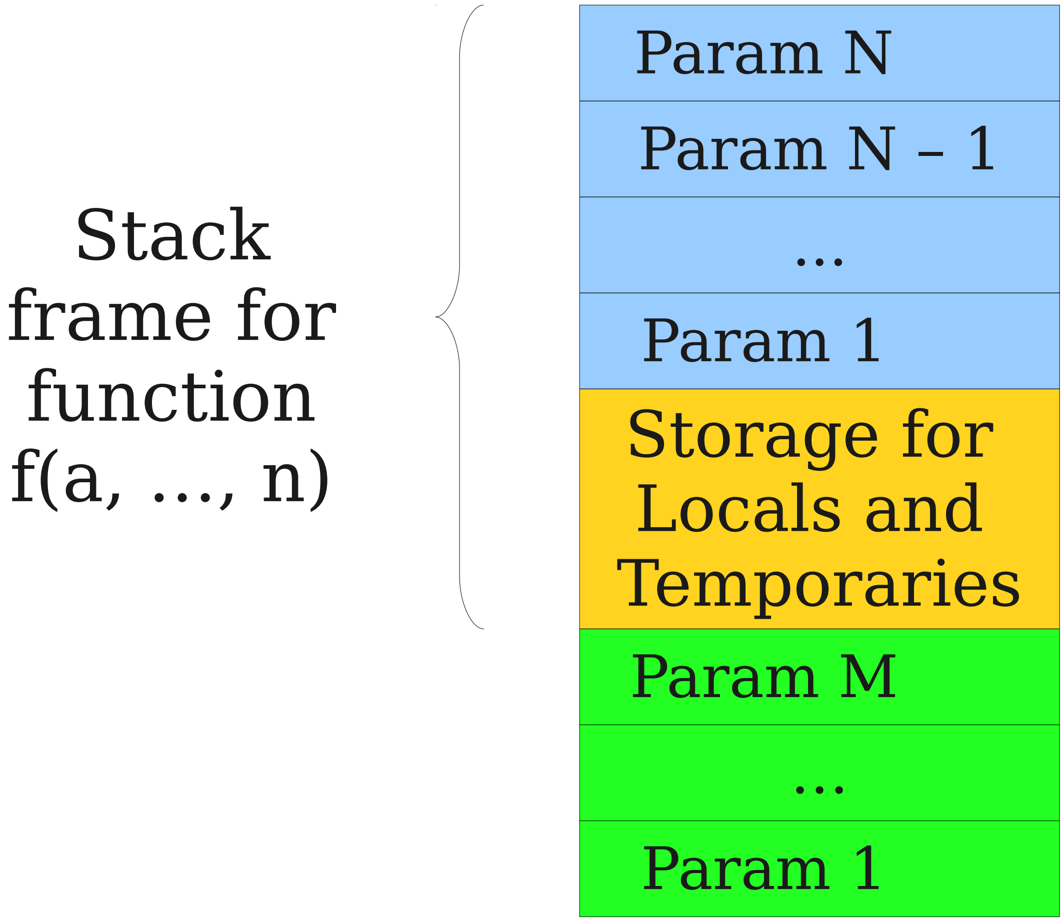
A Logical Decaf Stack Frame



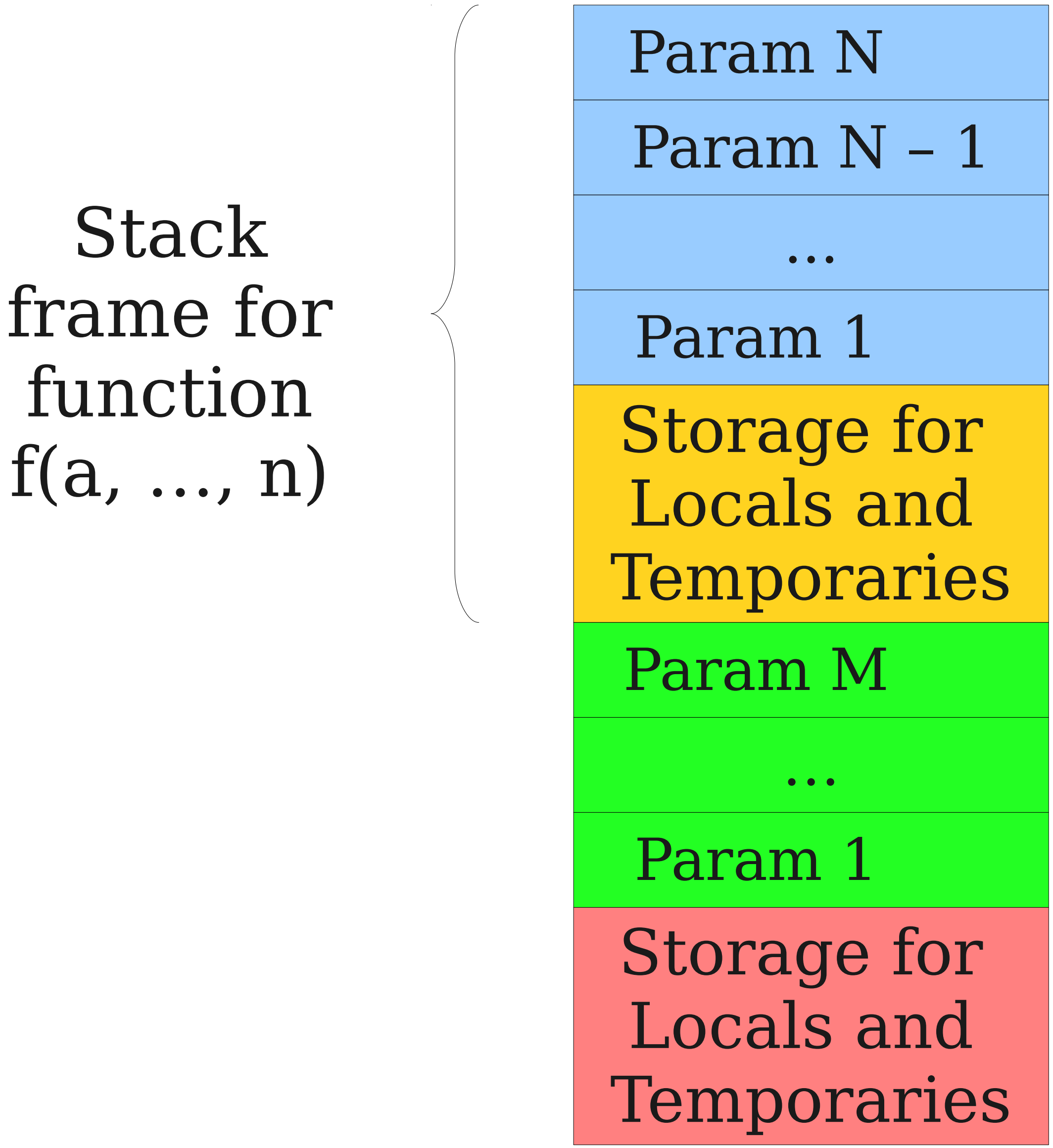
A Logical Decaf Stack Frame



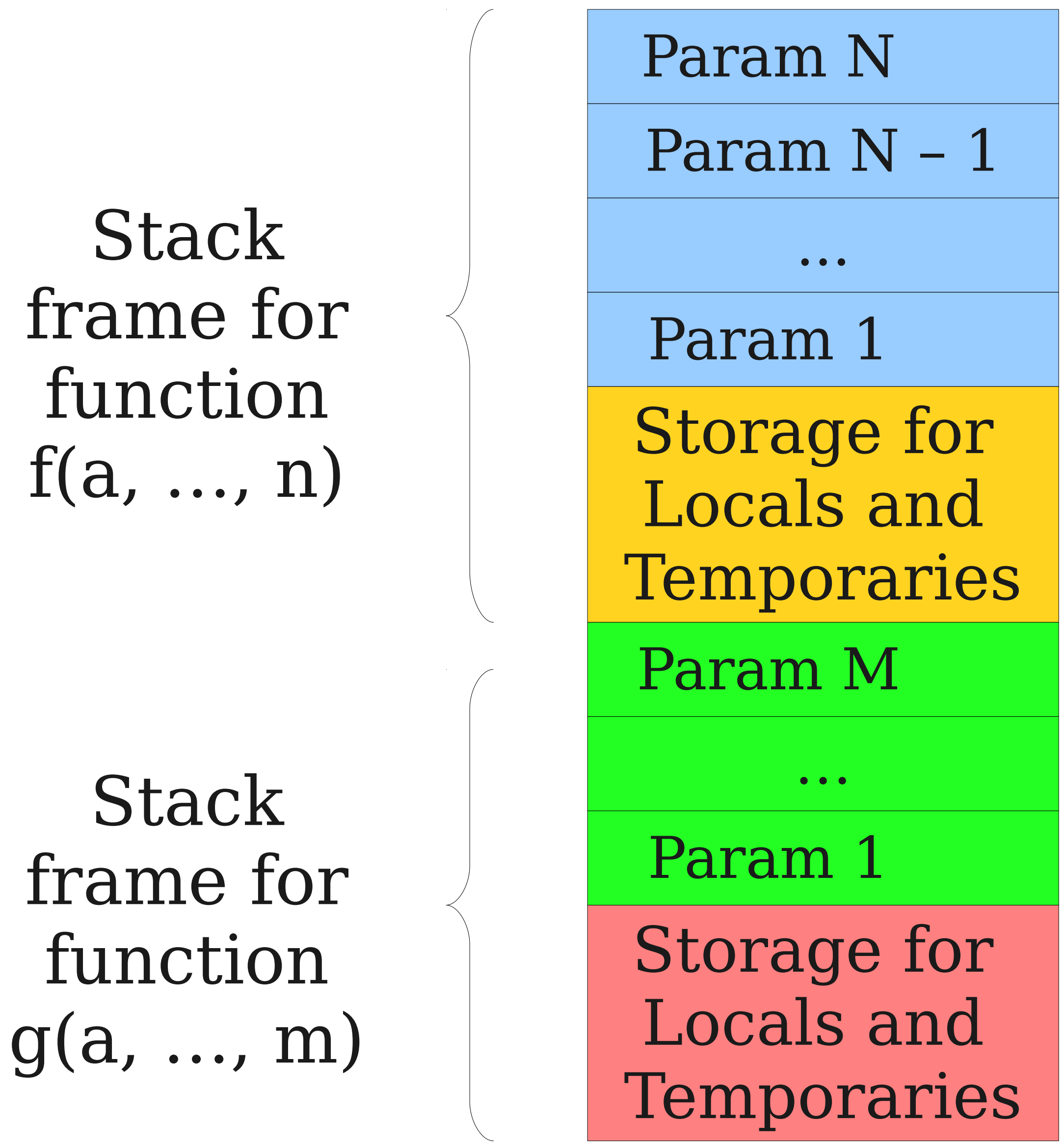
A Logical Decaf Stack Frame



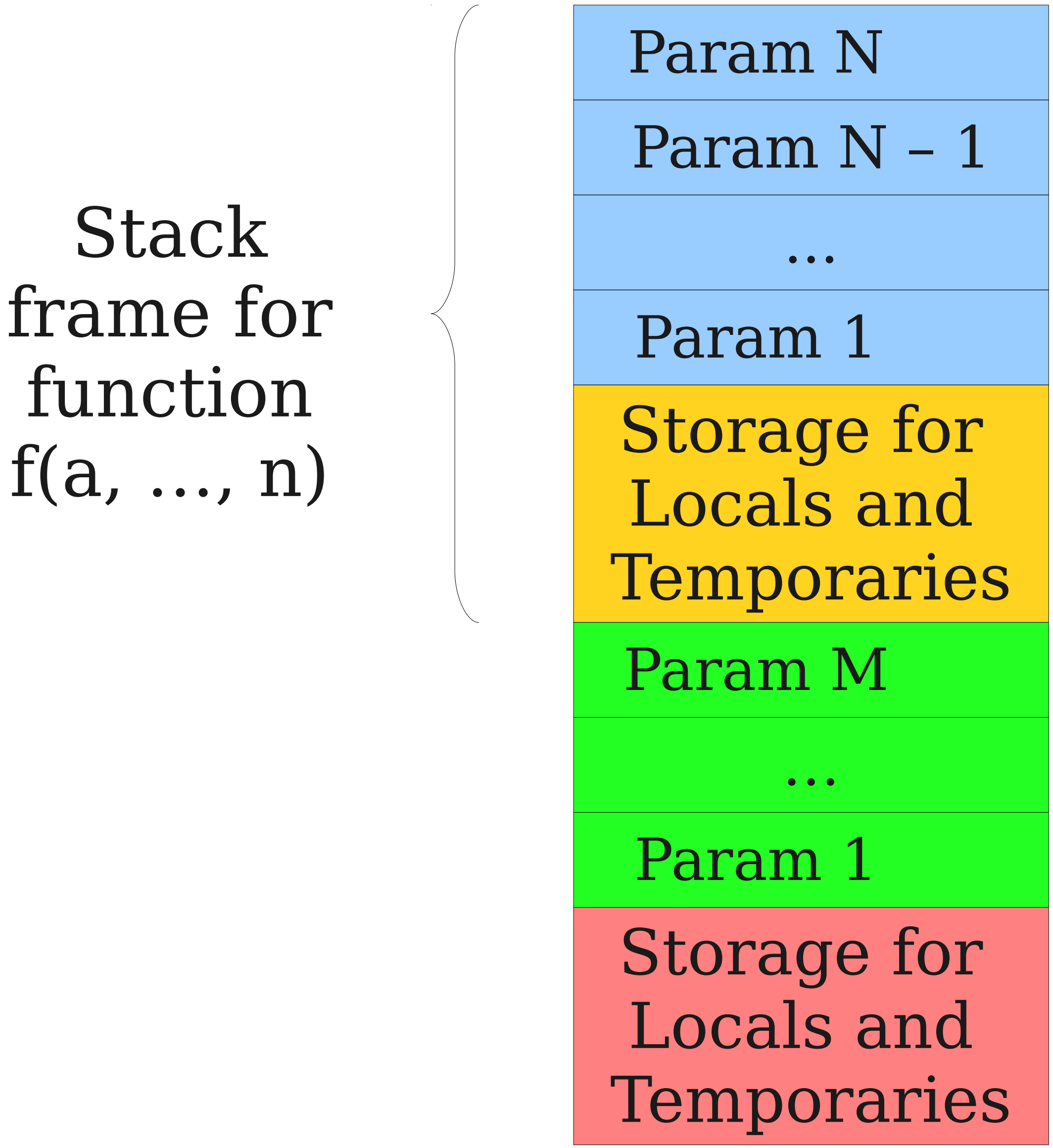
A Logical Decaf Stack Frame



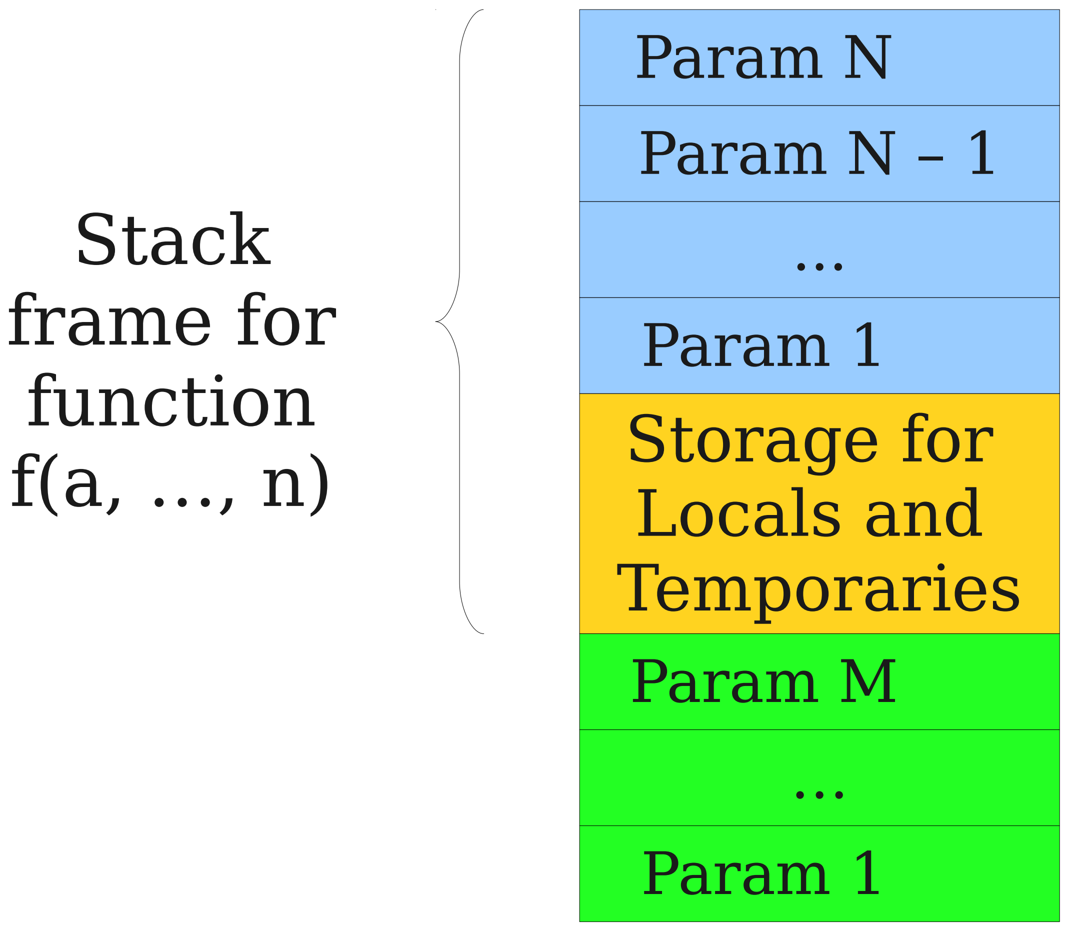
A Logical Decaf Stack Frame



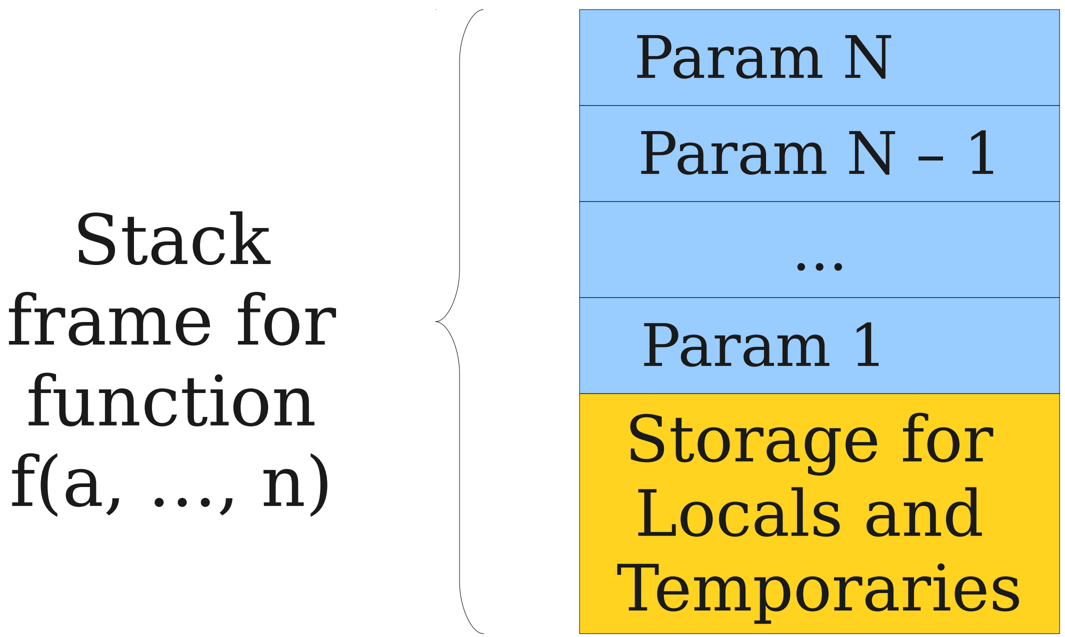
A Logical Decaf Stack Frame



A Logical Decaf Stack Frame



A Logical Decaf Stack Frame



Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
   SimpleFunction(137);
}
```

Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
   SimpleFunction(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;
```

Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
   SimpleFunction(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;
```

Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
   SimpleFunction(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;
```

Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
   SimpleFunction(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
```

Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
   SimpleFunction(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
```

Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
   SimpleFunction(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
```

Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
   SimpleFunction(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
```

Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}
```

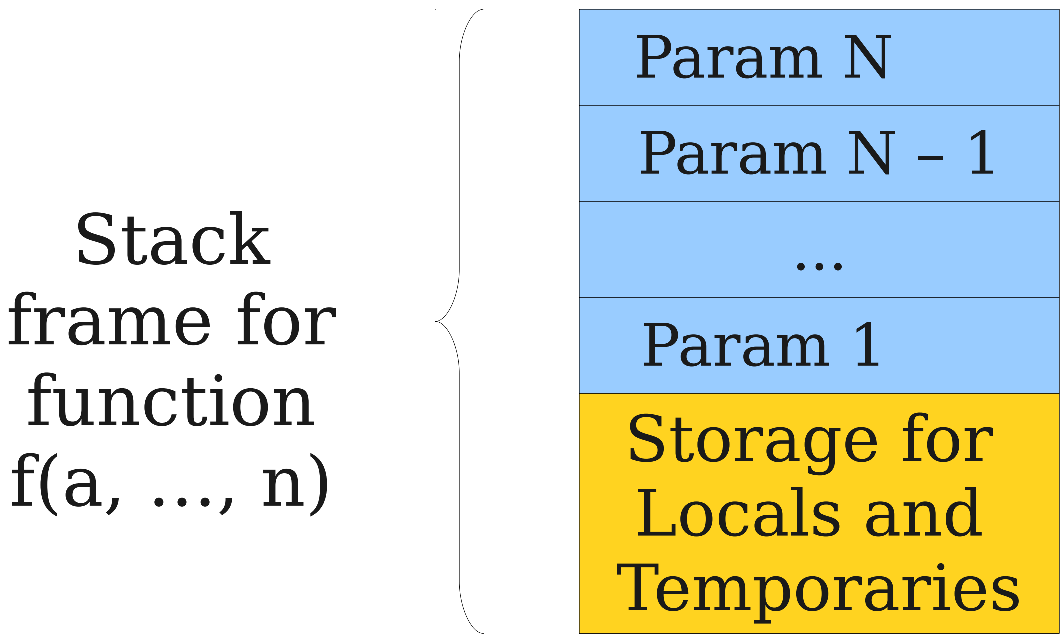
```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
```

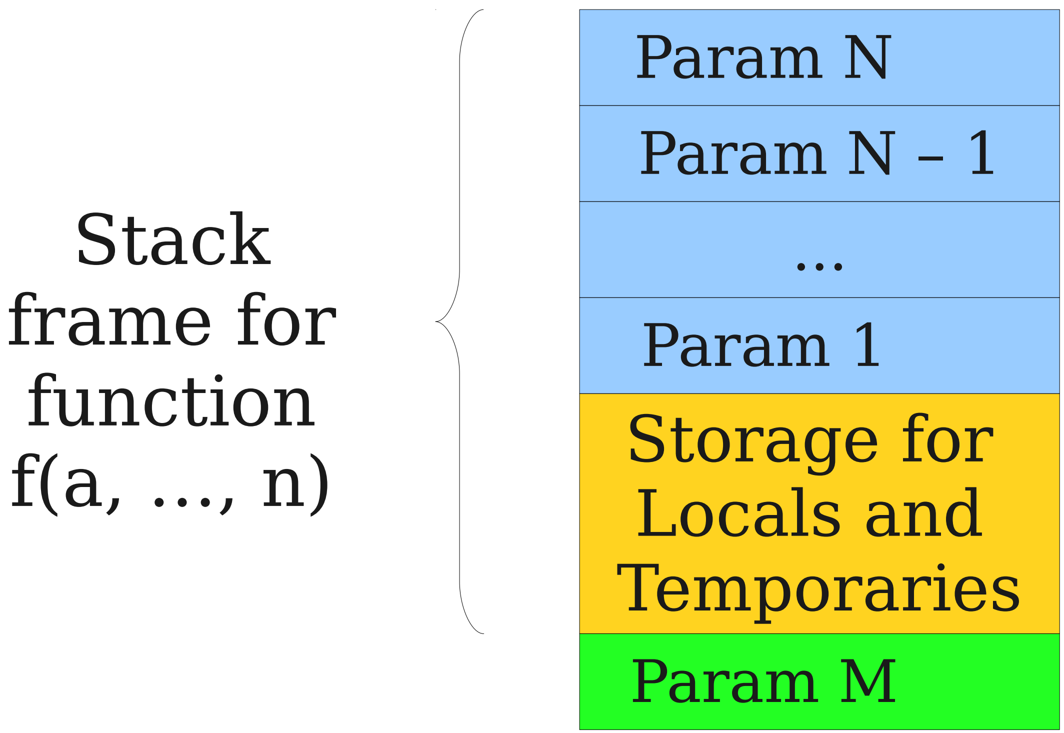
Stack Management in TAC

- The **`BeginFunc`** ***`N`***; instruction only needs to reserve room for local variables and temporaries.
- The **`EndFunc`**; instruction reclaims the room allocated with **`BeginFunc`** ***`N`***;
- A single parameter is pushed onto the stack by the caller using the **`PushParam`** ***`var`*** instruction.
- Space for parameters is reclaimed by the caller using the **`PopParams`** ***`N`***; instruction.
  - ***`N`*** is measured in *bytes*, not number of arguments.

A Logical Decaf Stack Frame

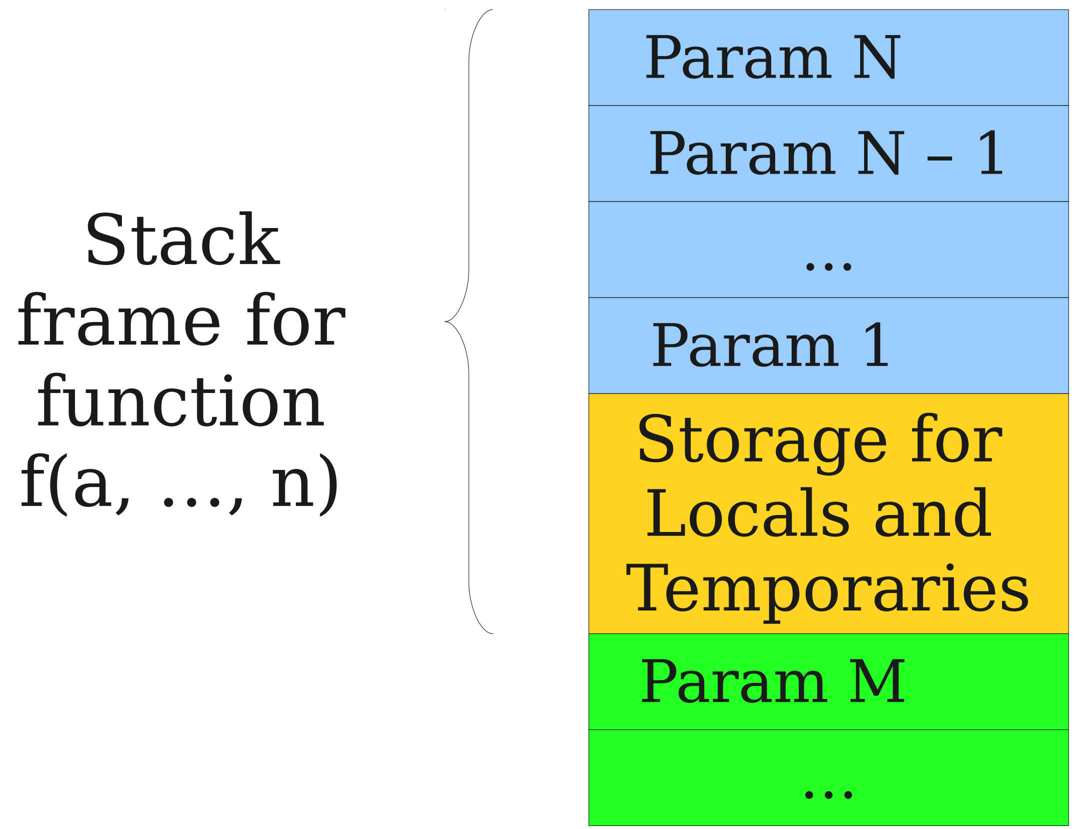


A Logical Decaf Stack Frame



`PushParam` ***`var`*****`;`**

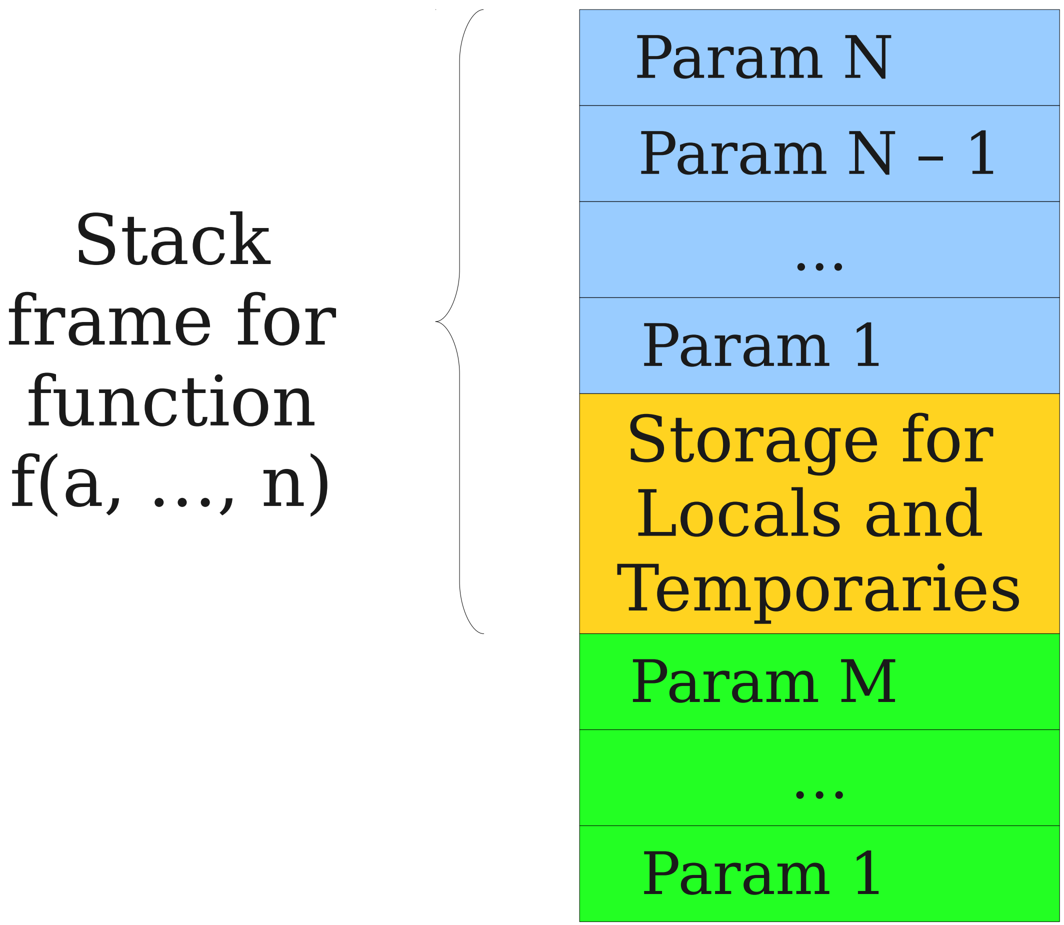
A Logical Decaf Stack Frame



PushParam *var*;

PushParam *var*;

A Logical Decaf Stack Frame

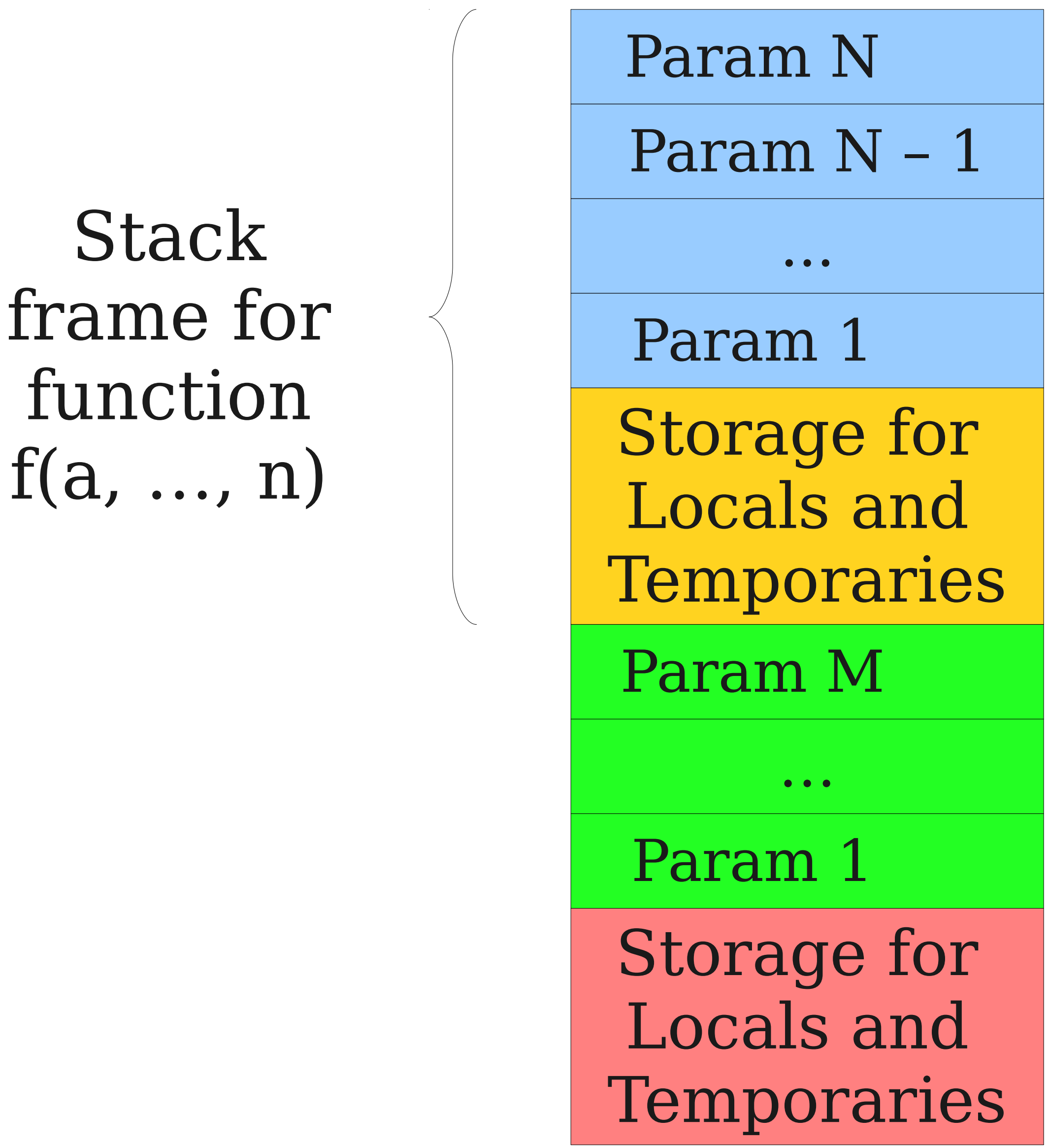


PushParam ***var*;**

PushParam ***var*;**

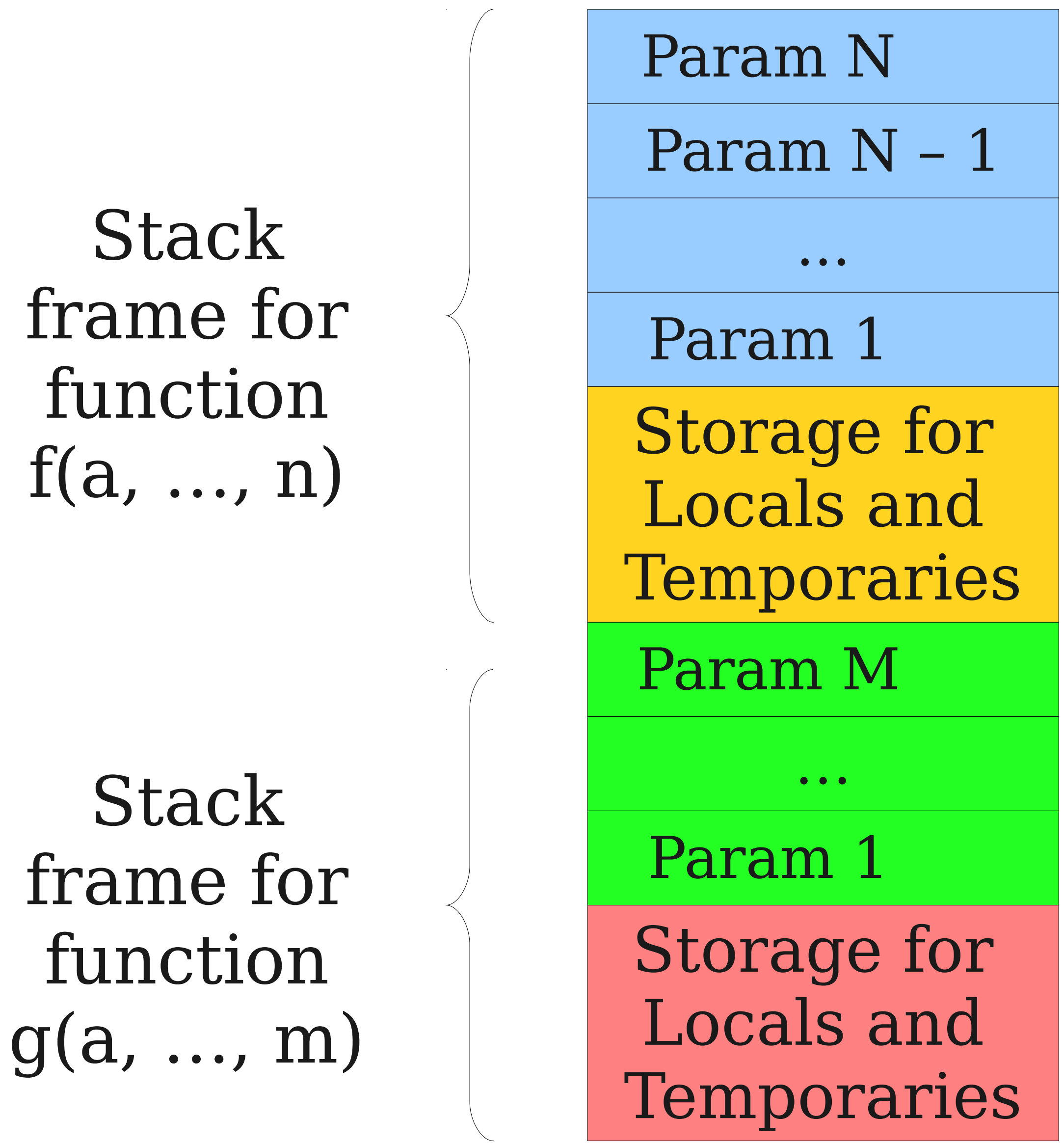
PushParam ***var*;**

A Logical Decaf Stack Frame



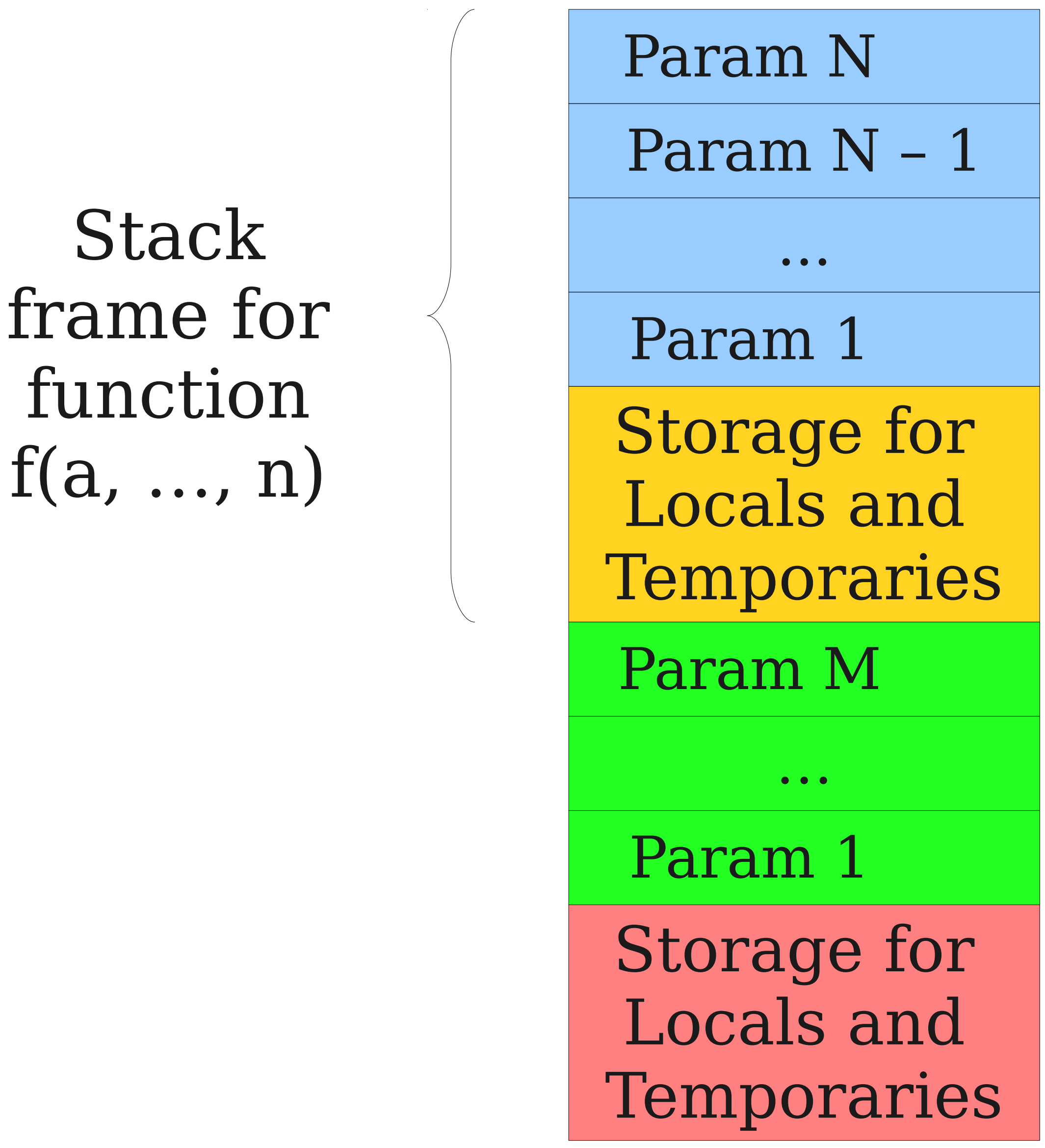

```
PushParam var;
PushParam var;
PushParam var;
BeginFunc N;
```

A Logical Decaf Stack Frame

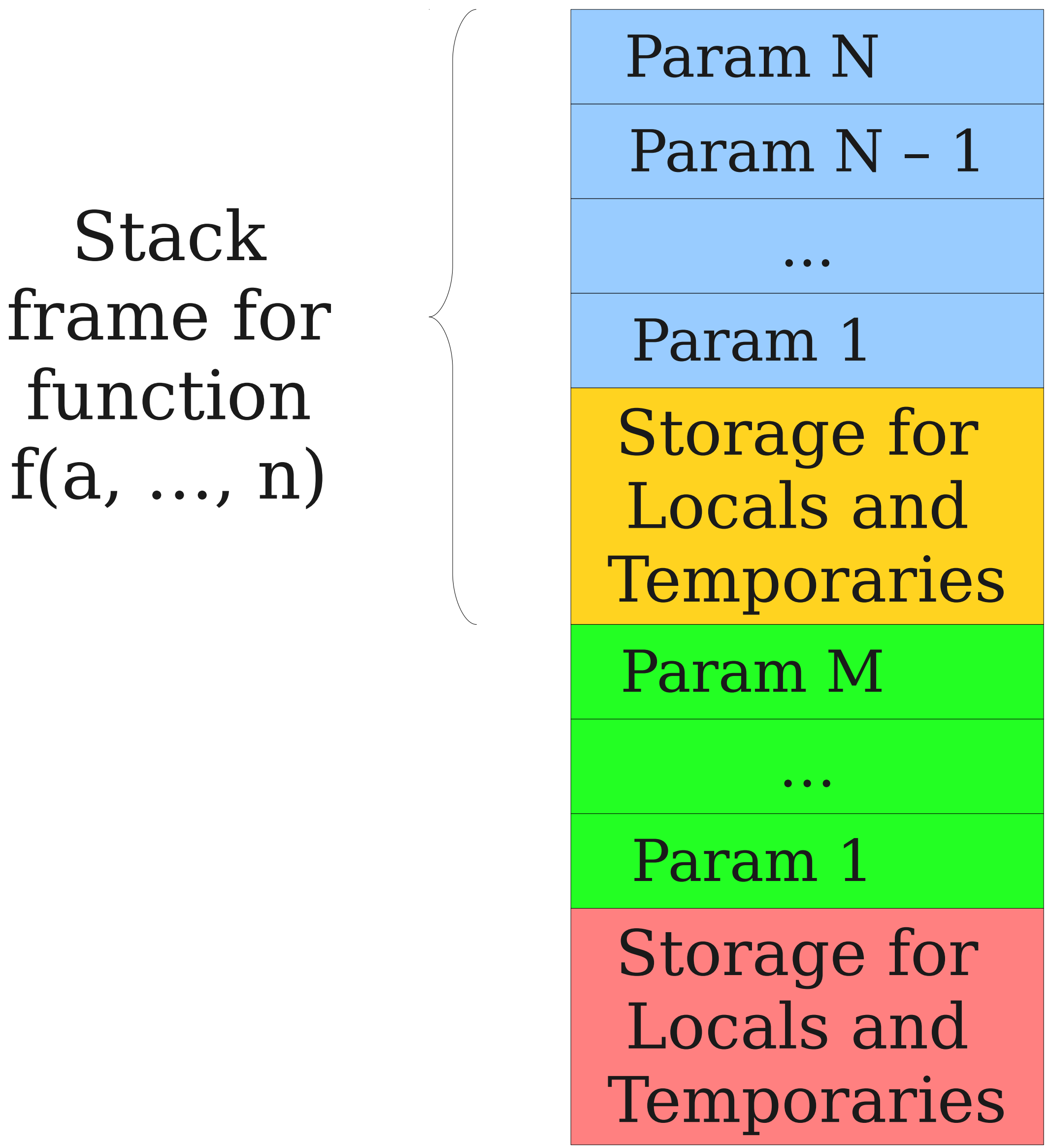



```
PushParam var;
PushParam var;
PushParam var;
BeginFunc N;
```

A Logical Decaf Stack Frame

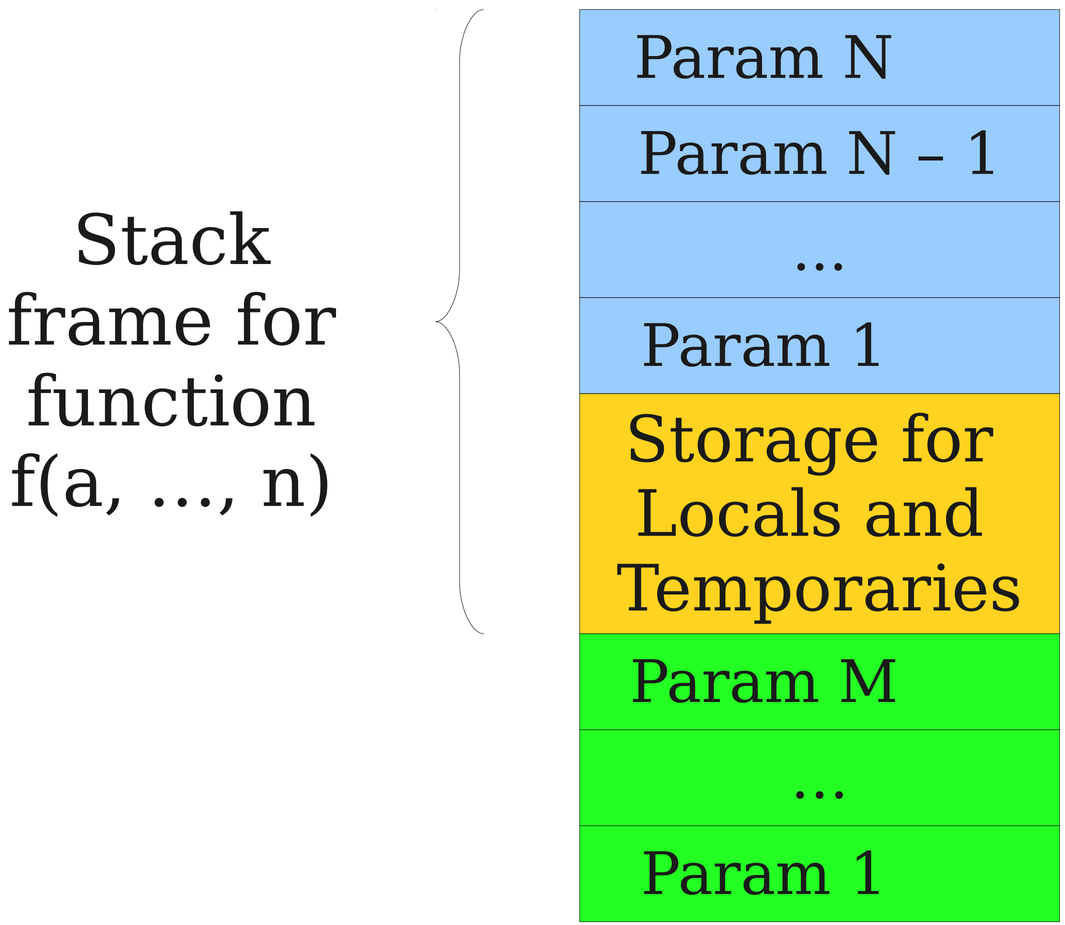


A Logical Decaf Stack Frame

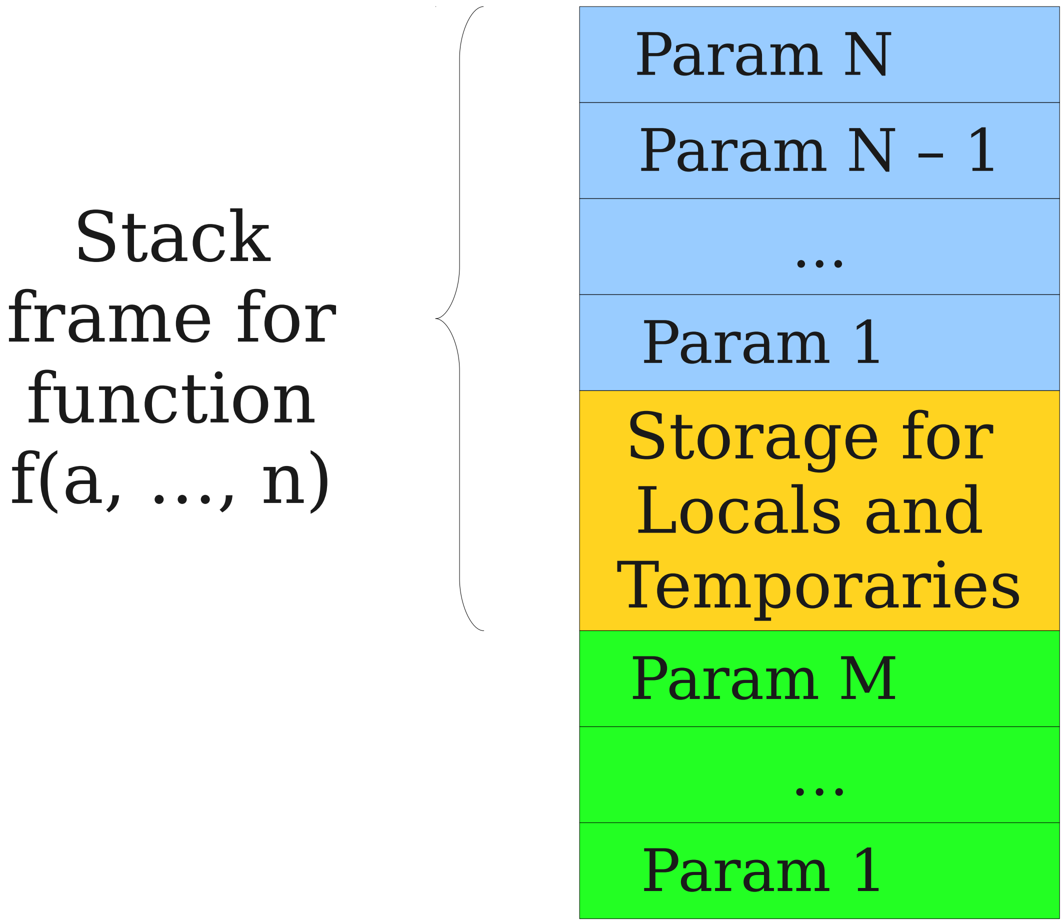


`EndFunc;`

A Logical Decaf Stack Frame

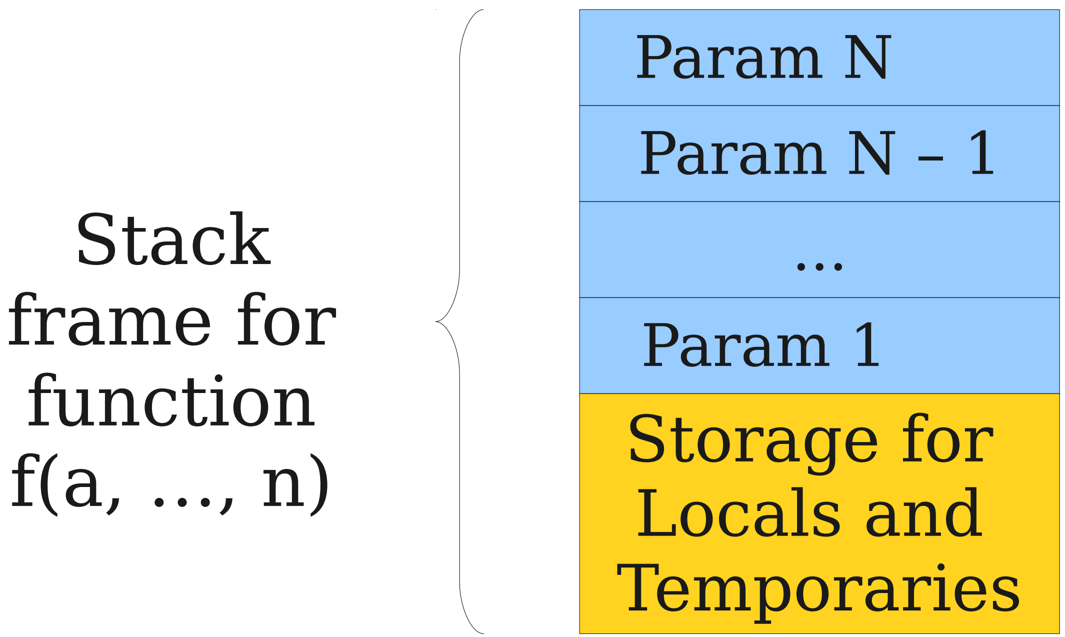


A Logical Decaf Stack Frame



PopParams *N*;

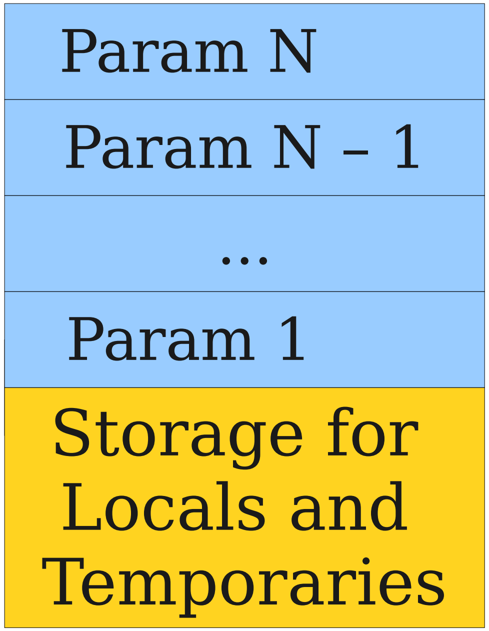
A Logical Decaf Stack Frame



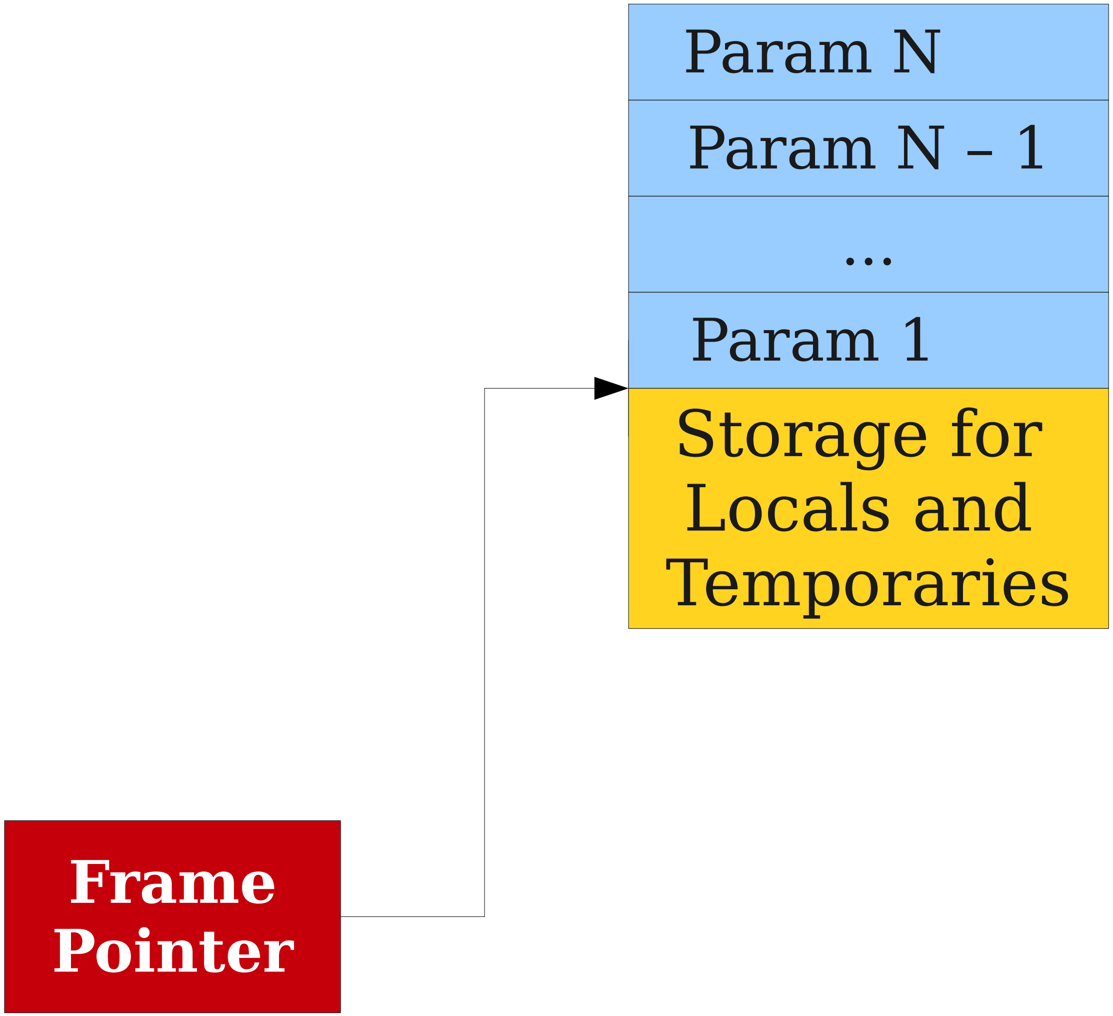
Storage Allocation

- As described so far, TAC does not specify where variables and temporaries are stored.
- For the final programming project, you will need to tell the code generator where each variable should be stored.
- This normally would be handled during code generation, but Just For Fun we thought you should have some experience handling this. ☺

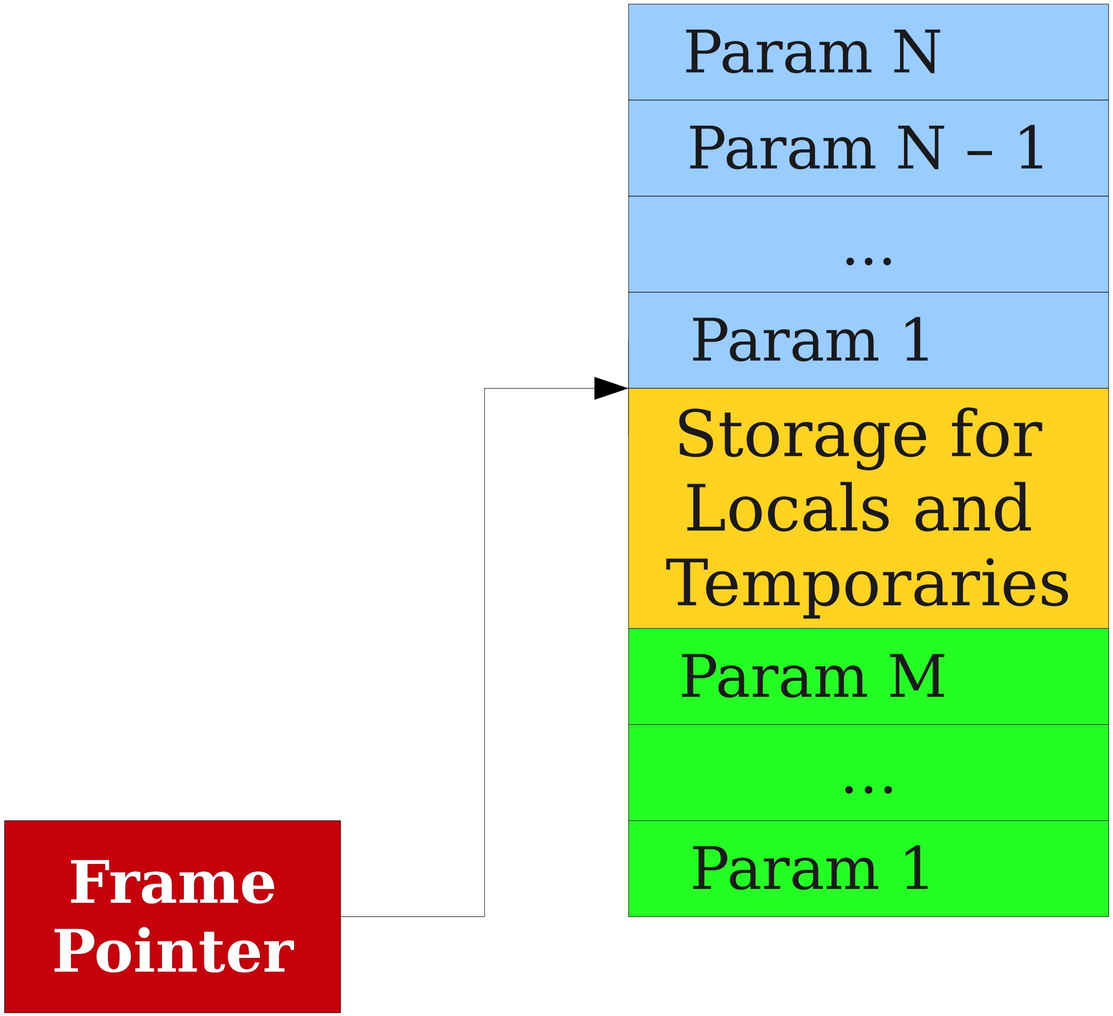
The Frame Pointer



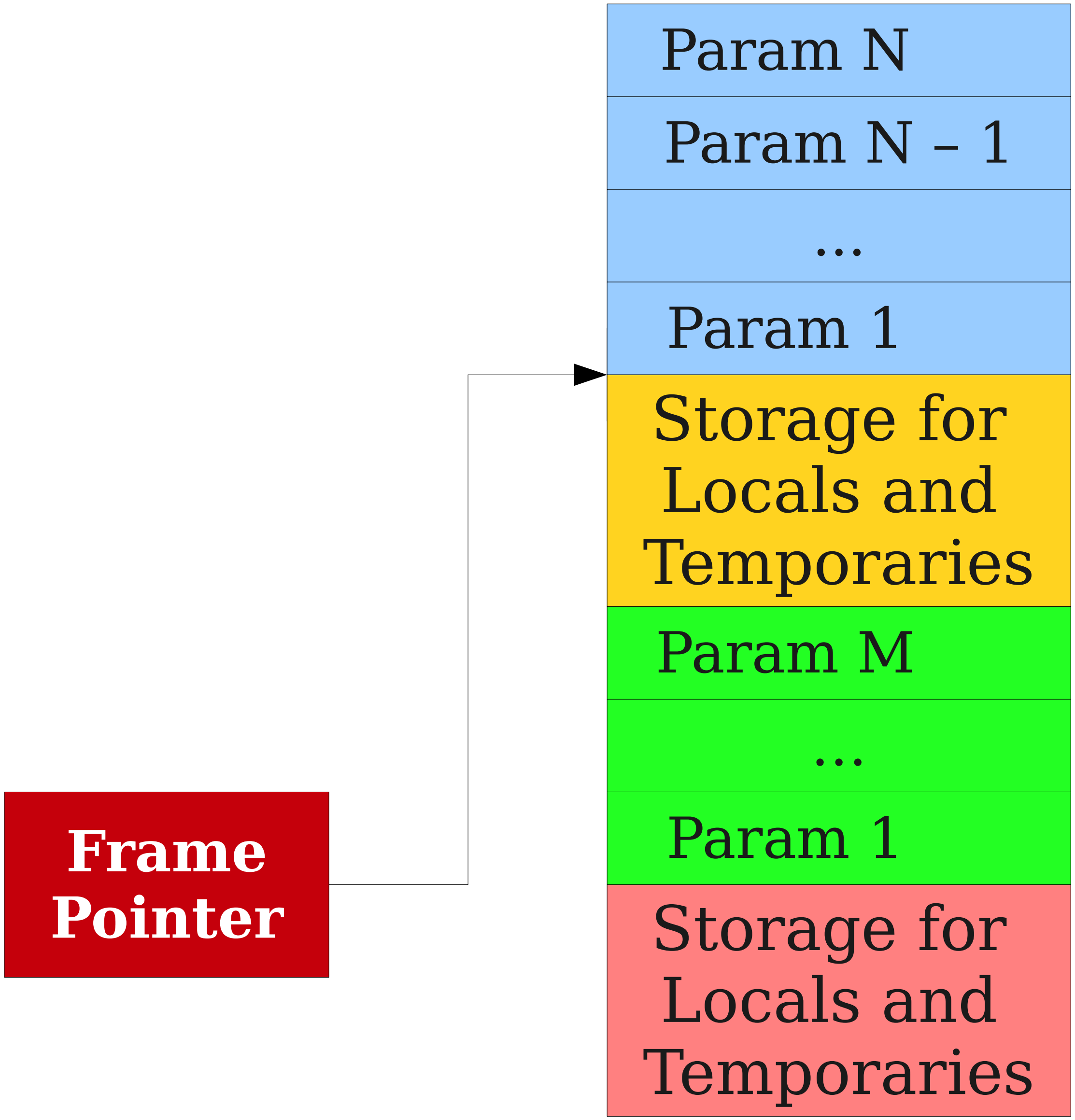
The Frame Pointer



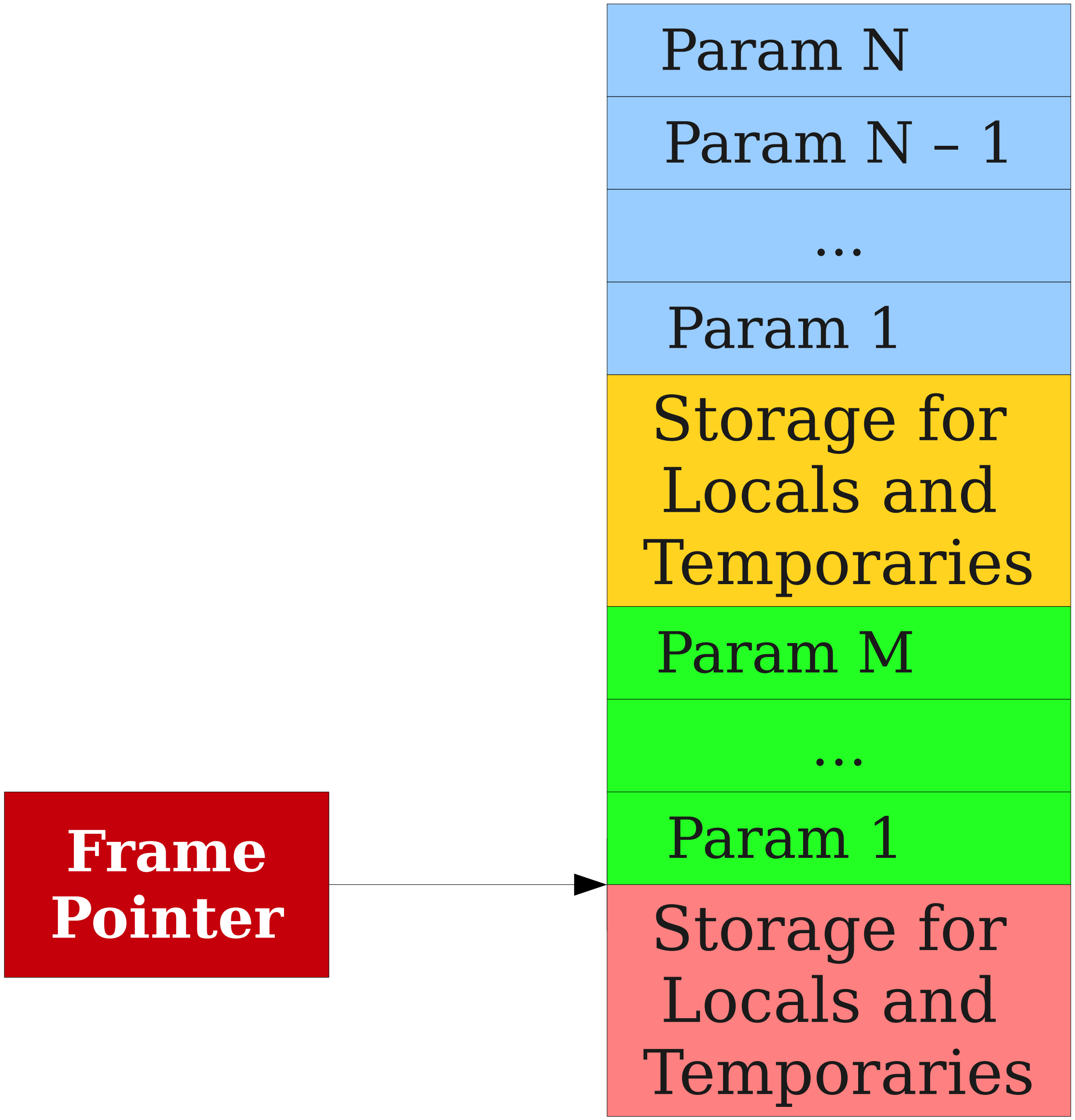
The Frame Pointer



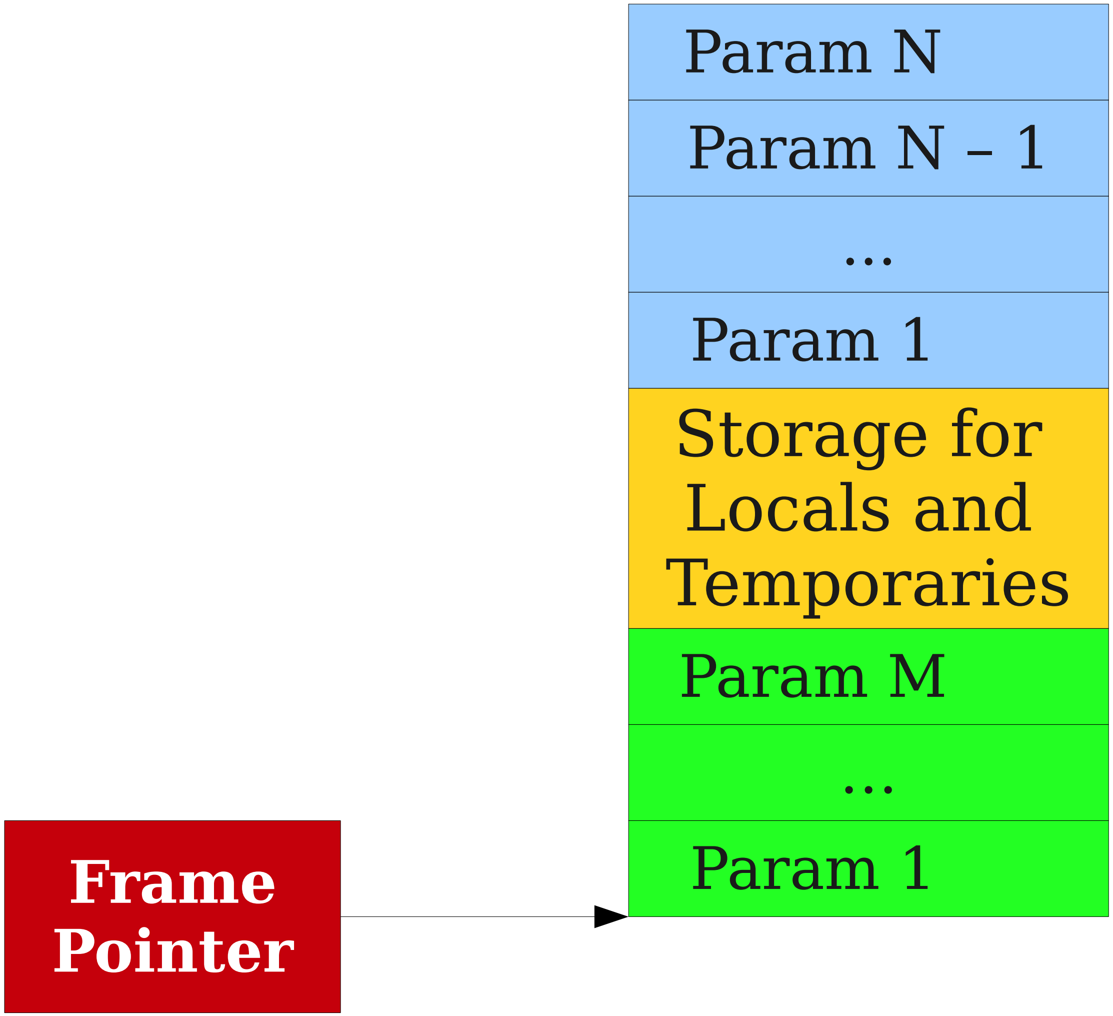
The Frame Pointer



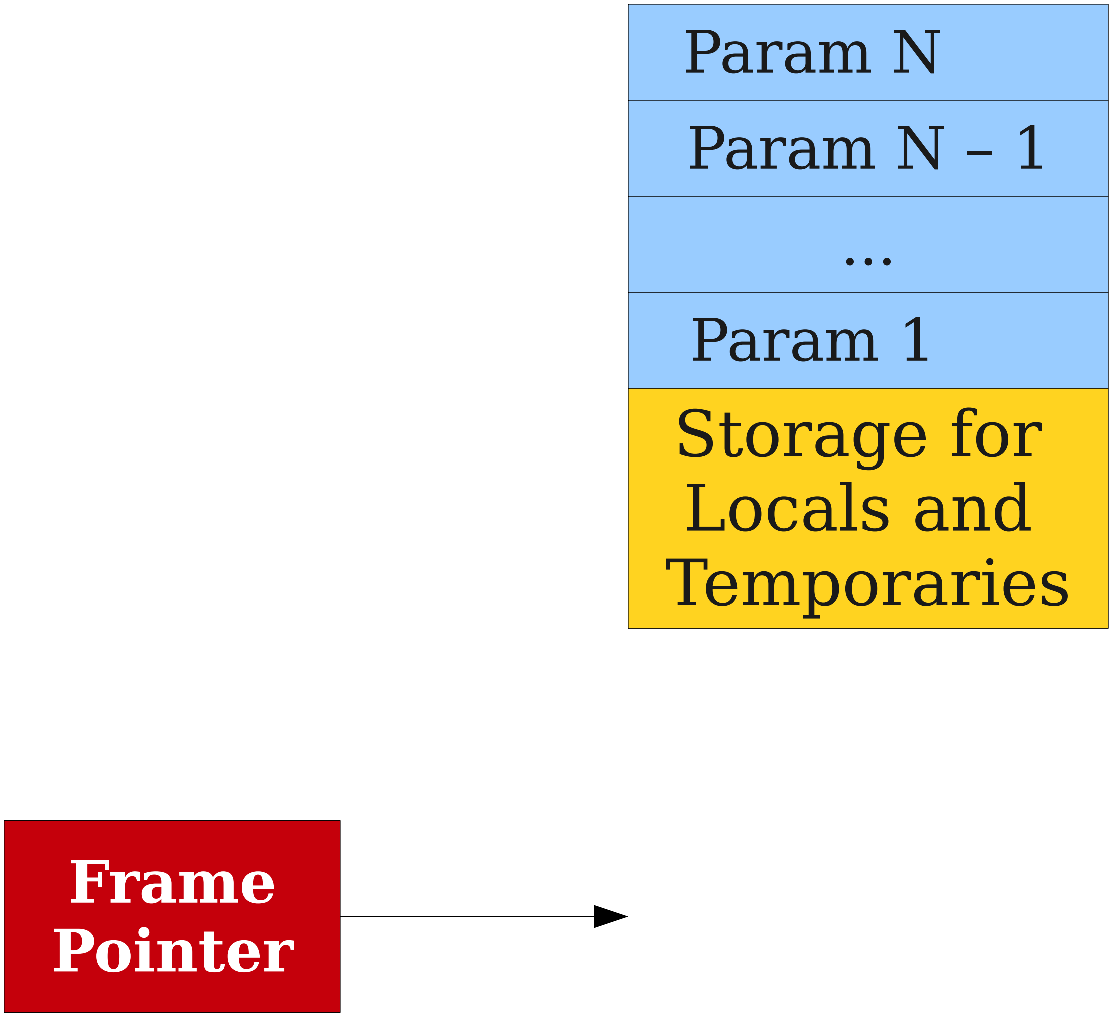
The Frame Pointer



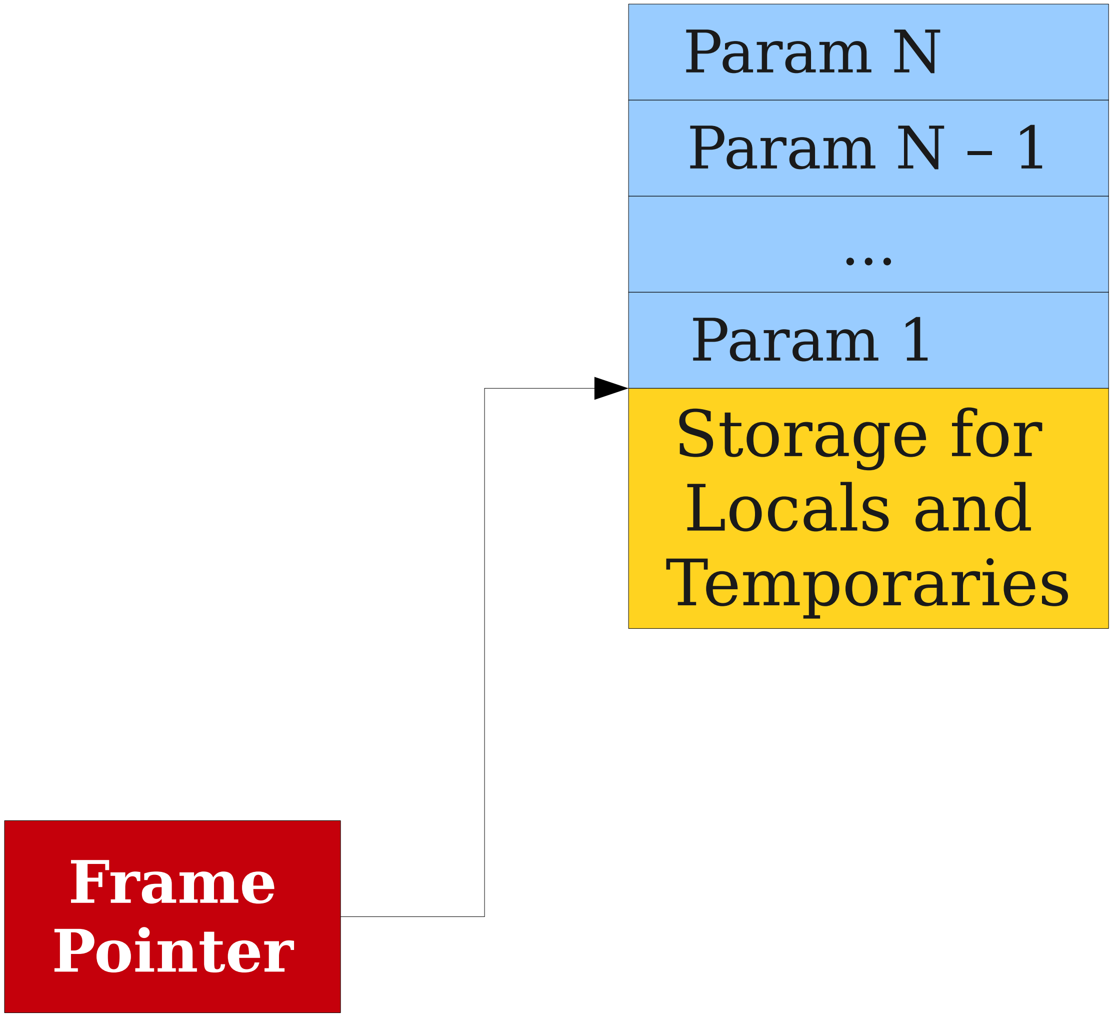
The Frame Pointer



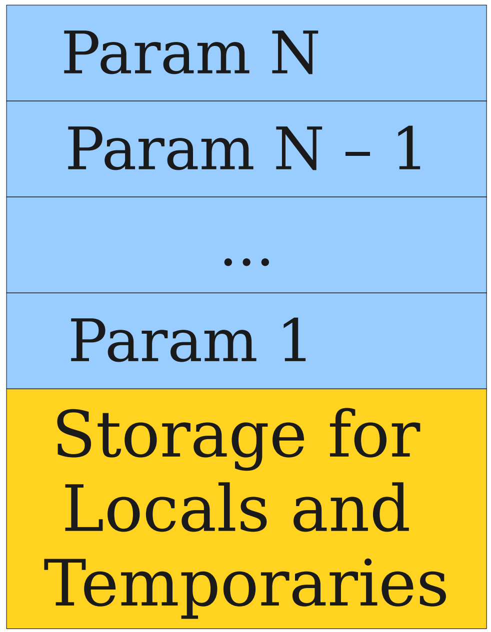
The Frame Pointer



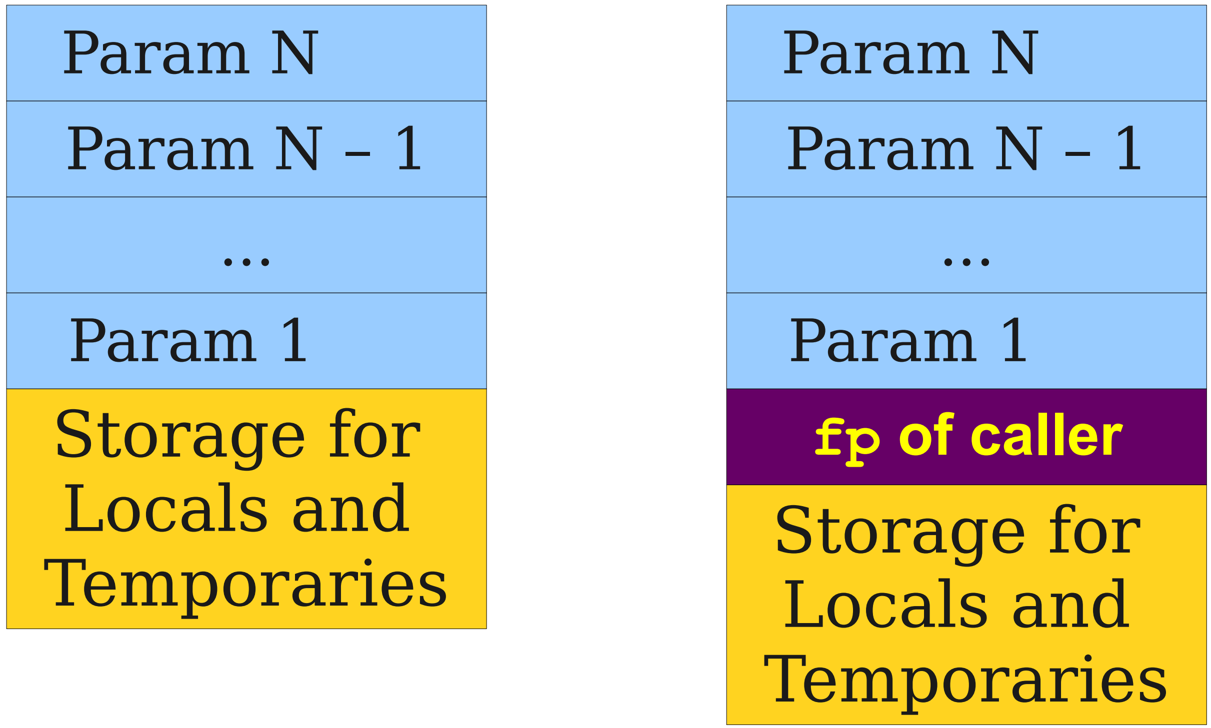
The Frame Pointer



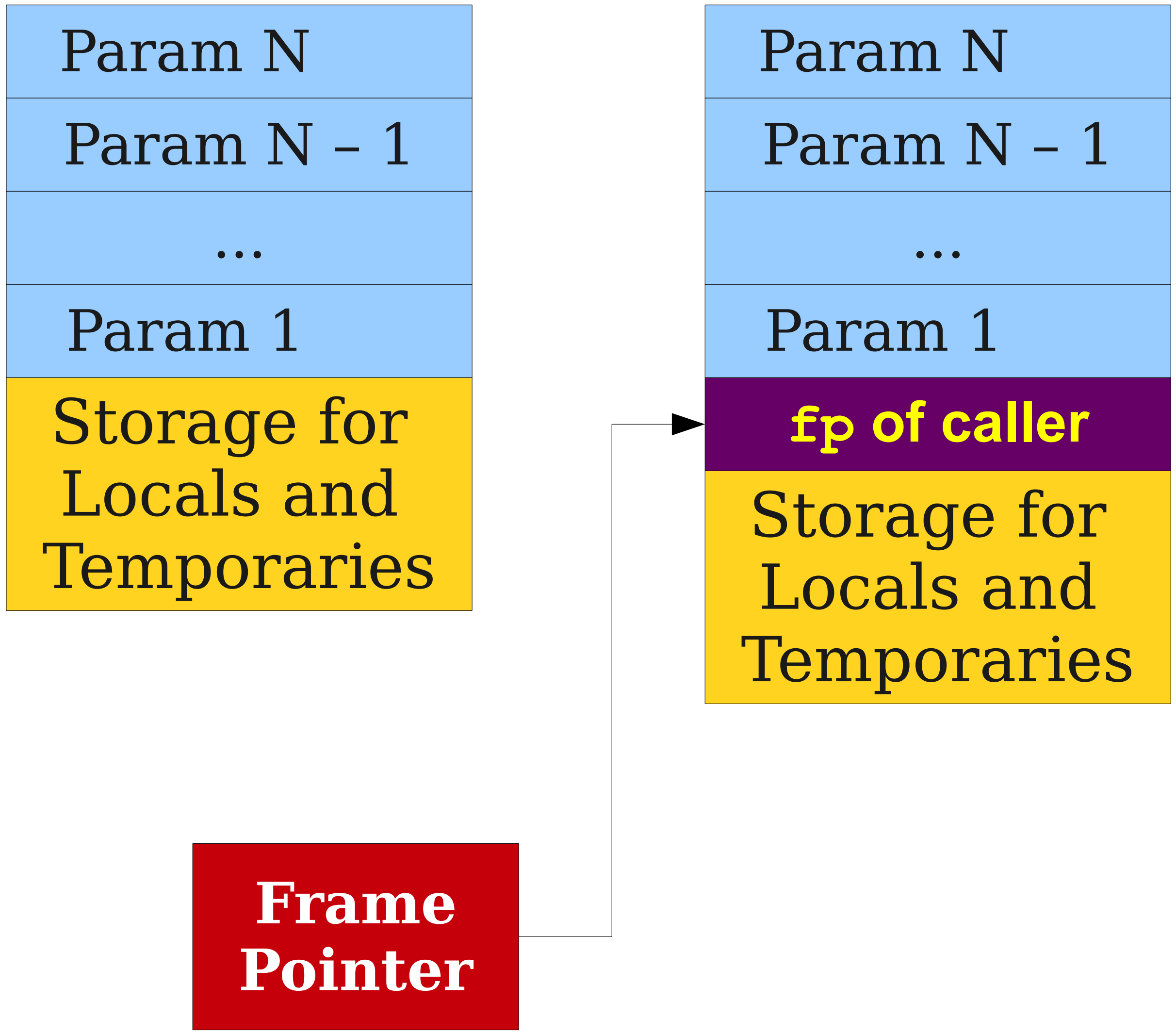
Logical vs Physical Stack Frames



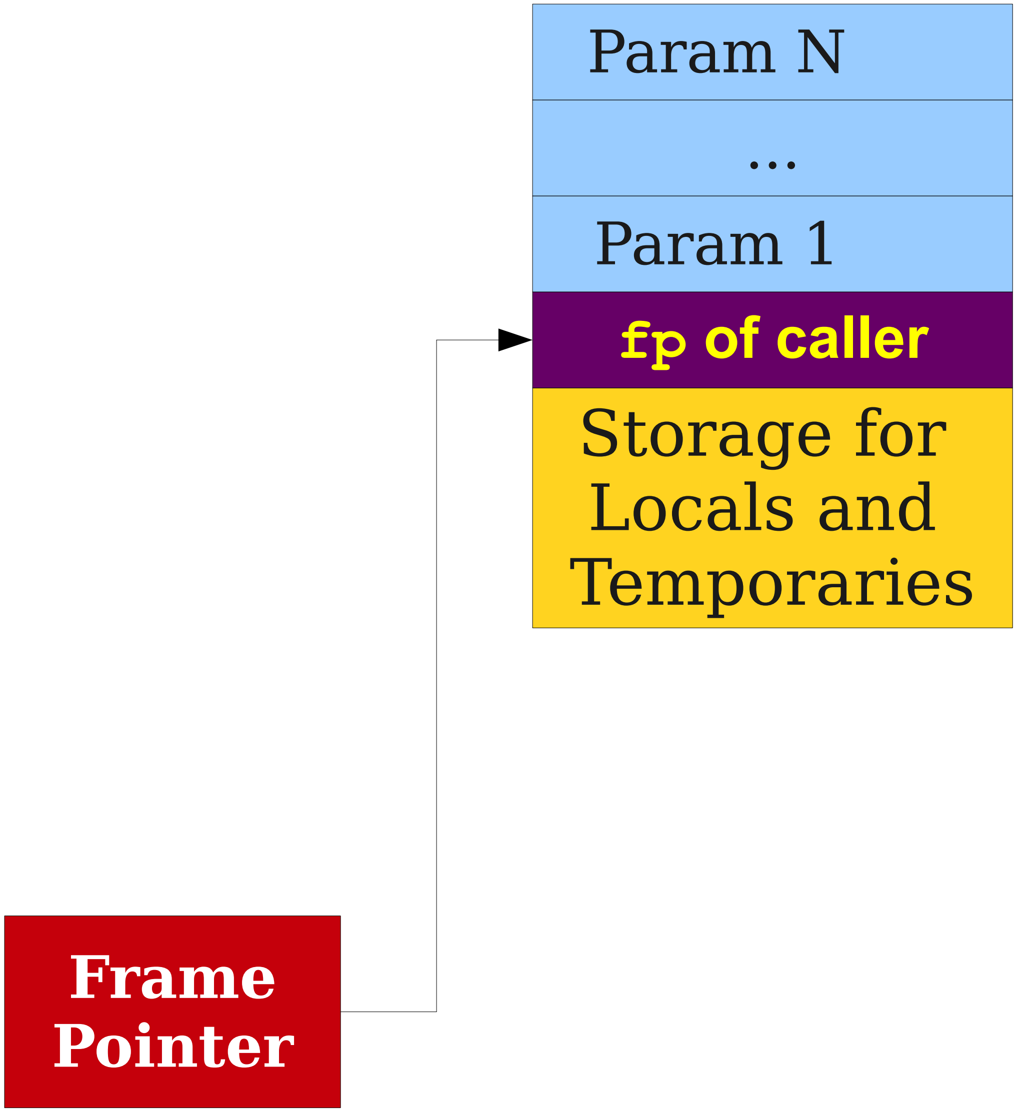
Logical vs Physical Stack Frames



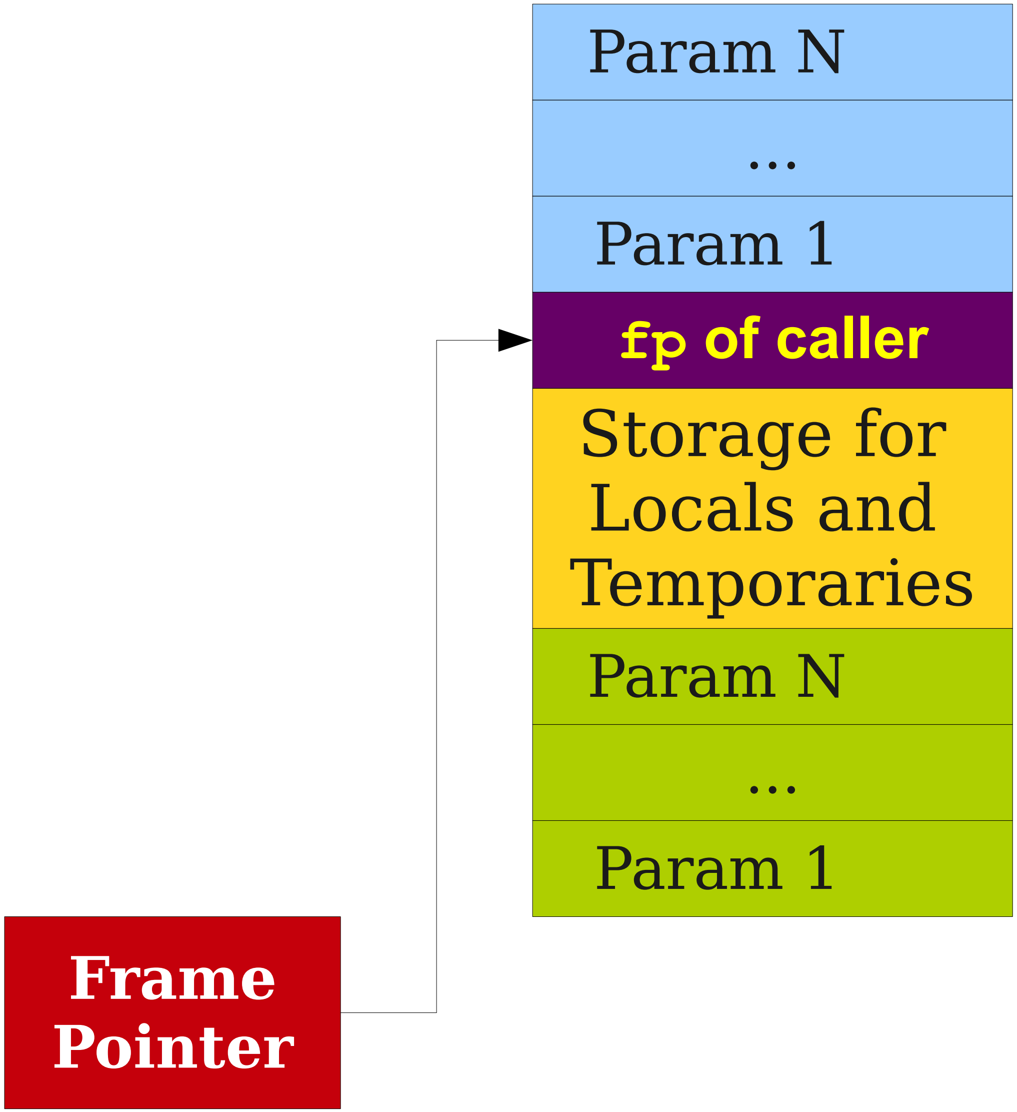
Logical vs Physical Stack Frames



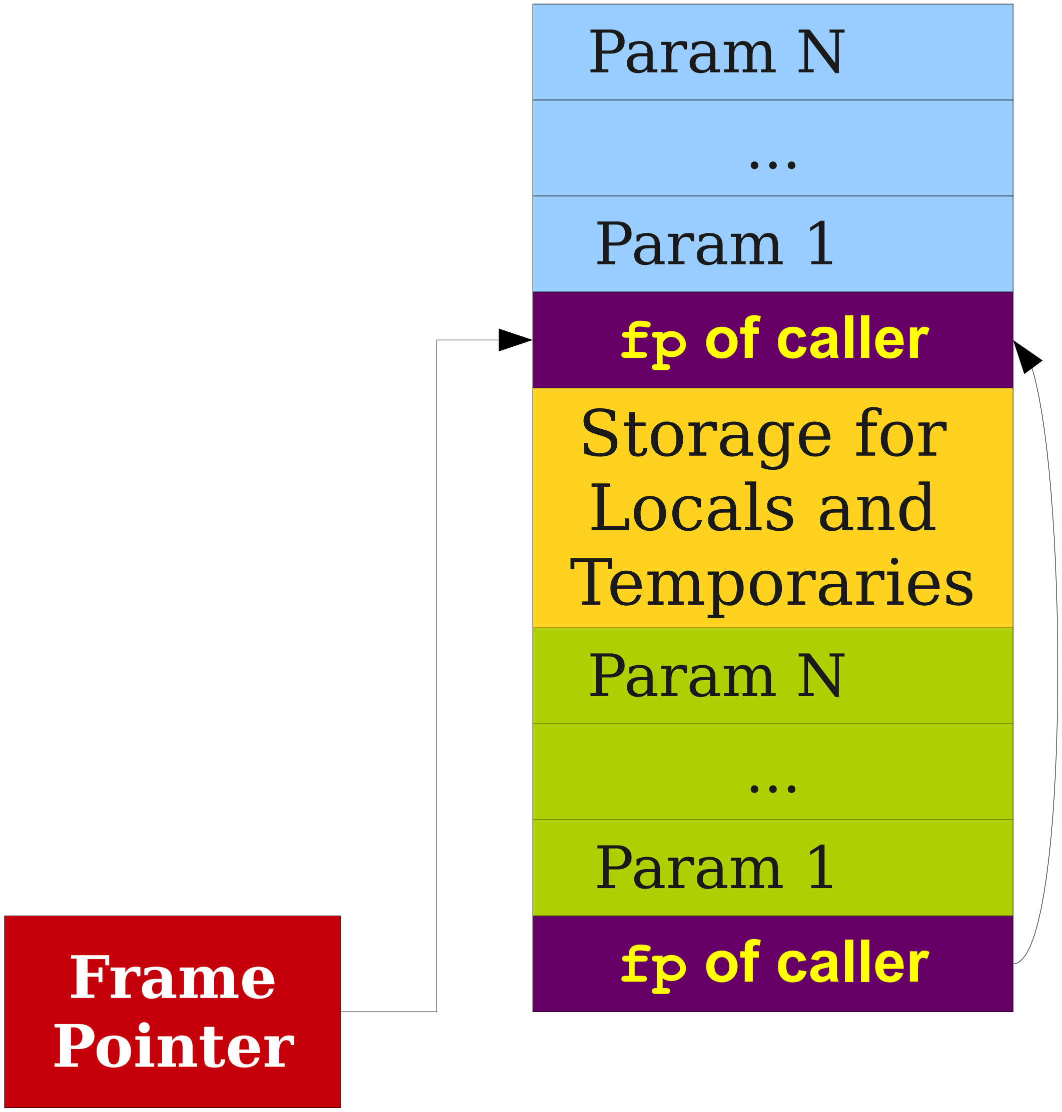
(Mostly) Physical Stack Frames



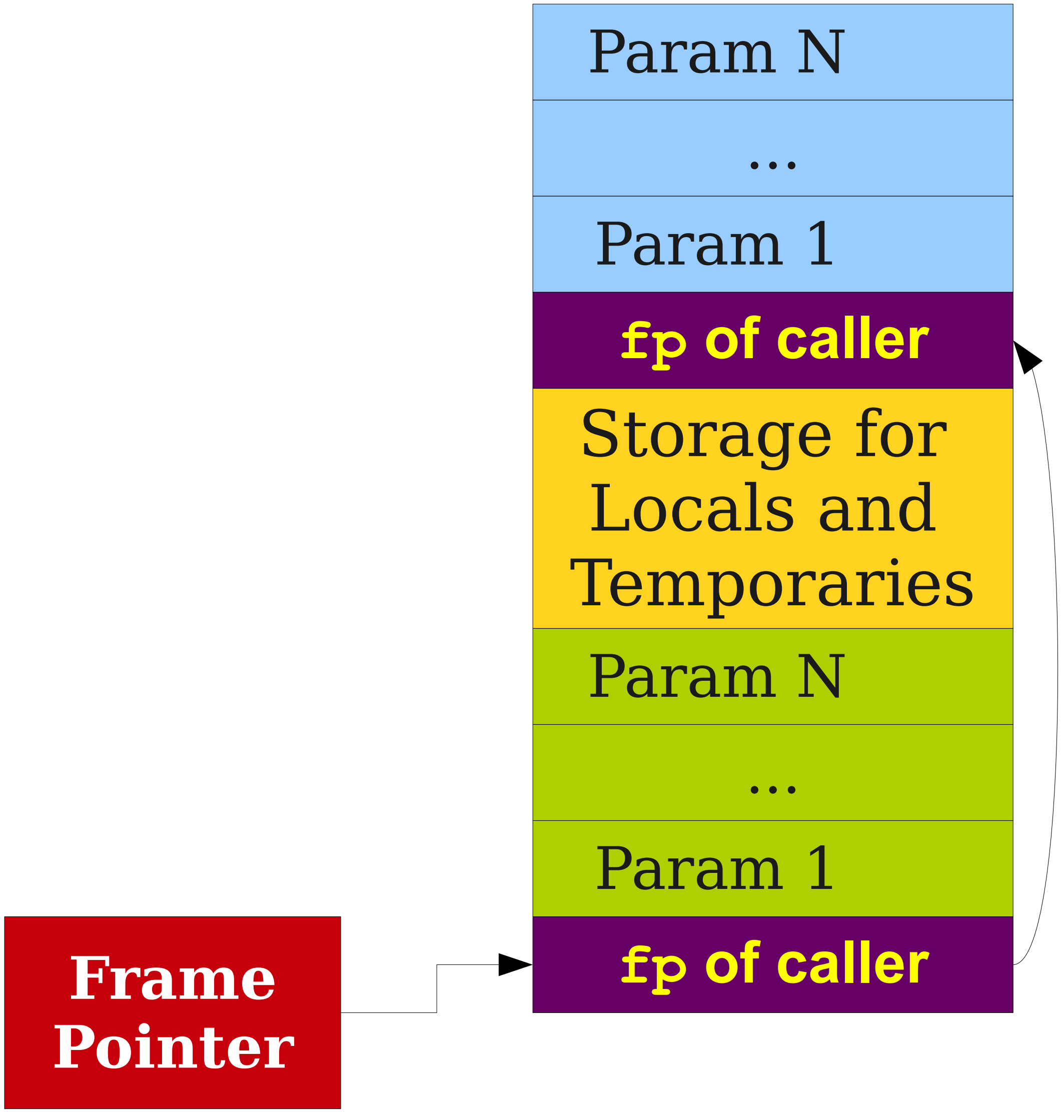
(Mostly) Physical Stack Frames



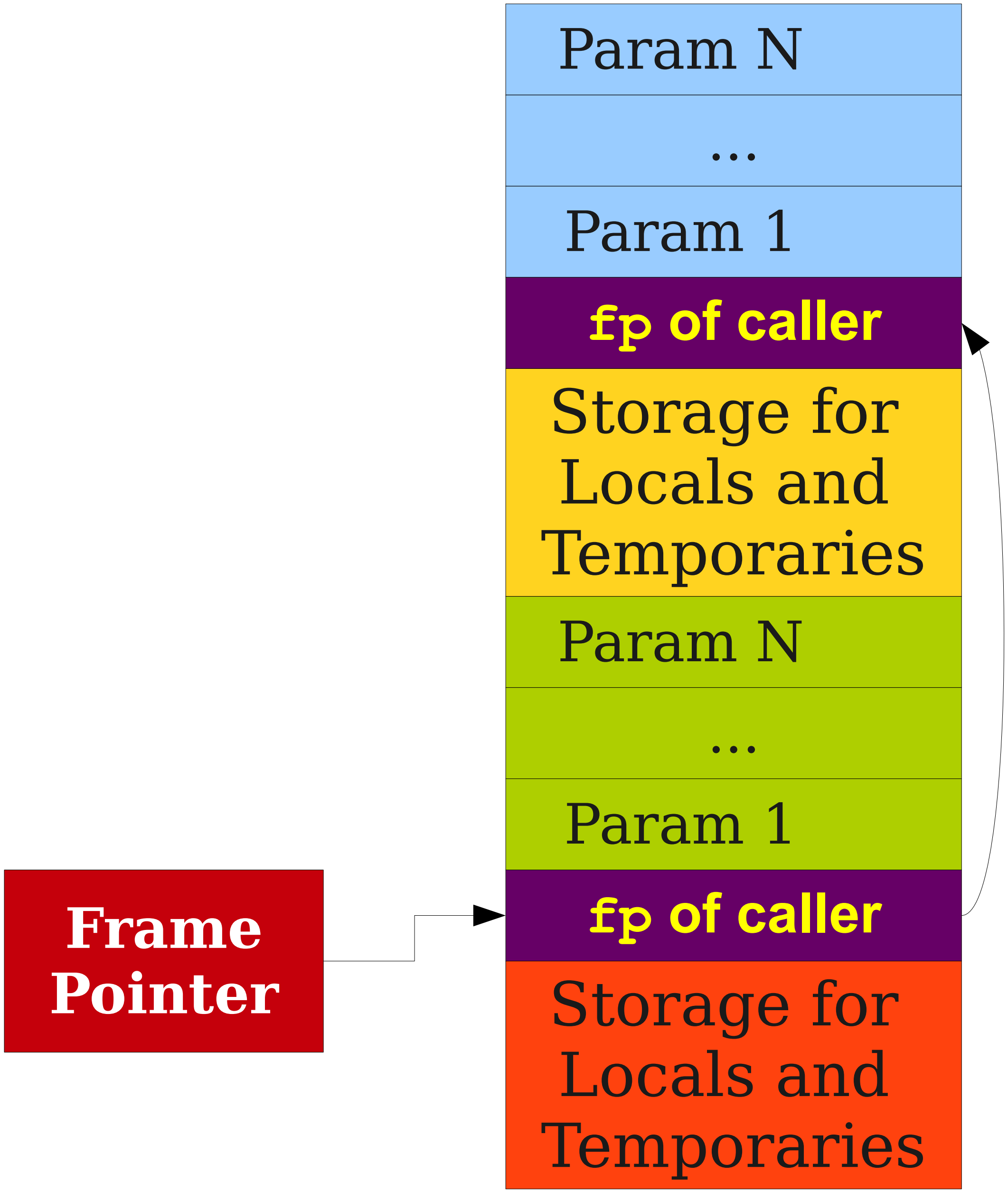
(Mostly) Physical Stack Frames



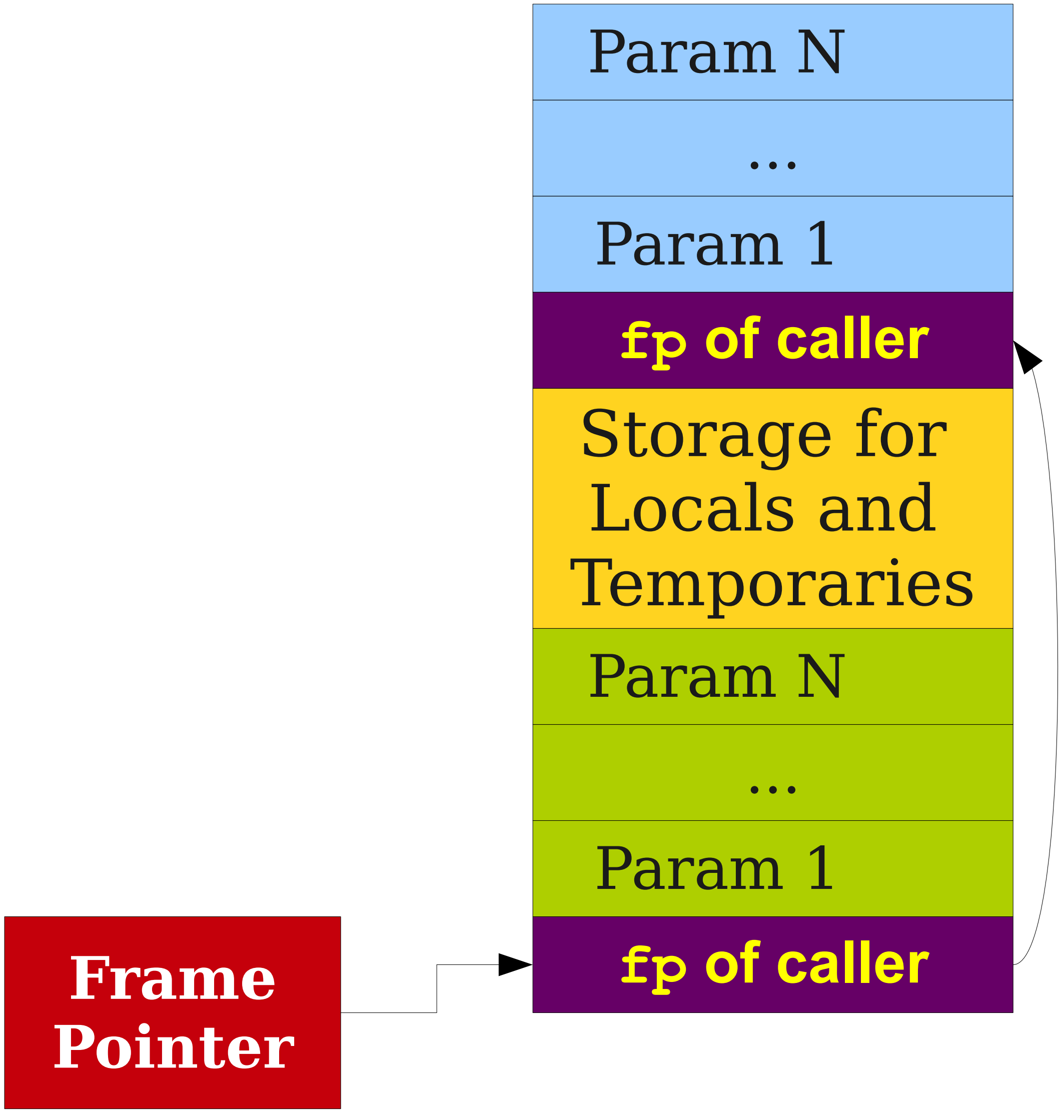
(Mostly) Physical Stack Frames



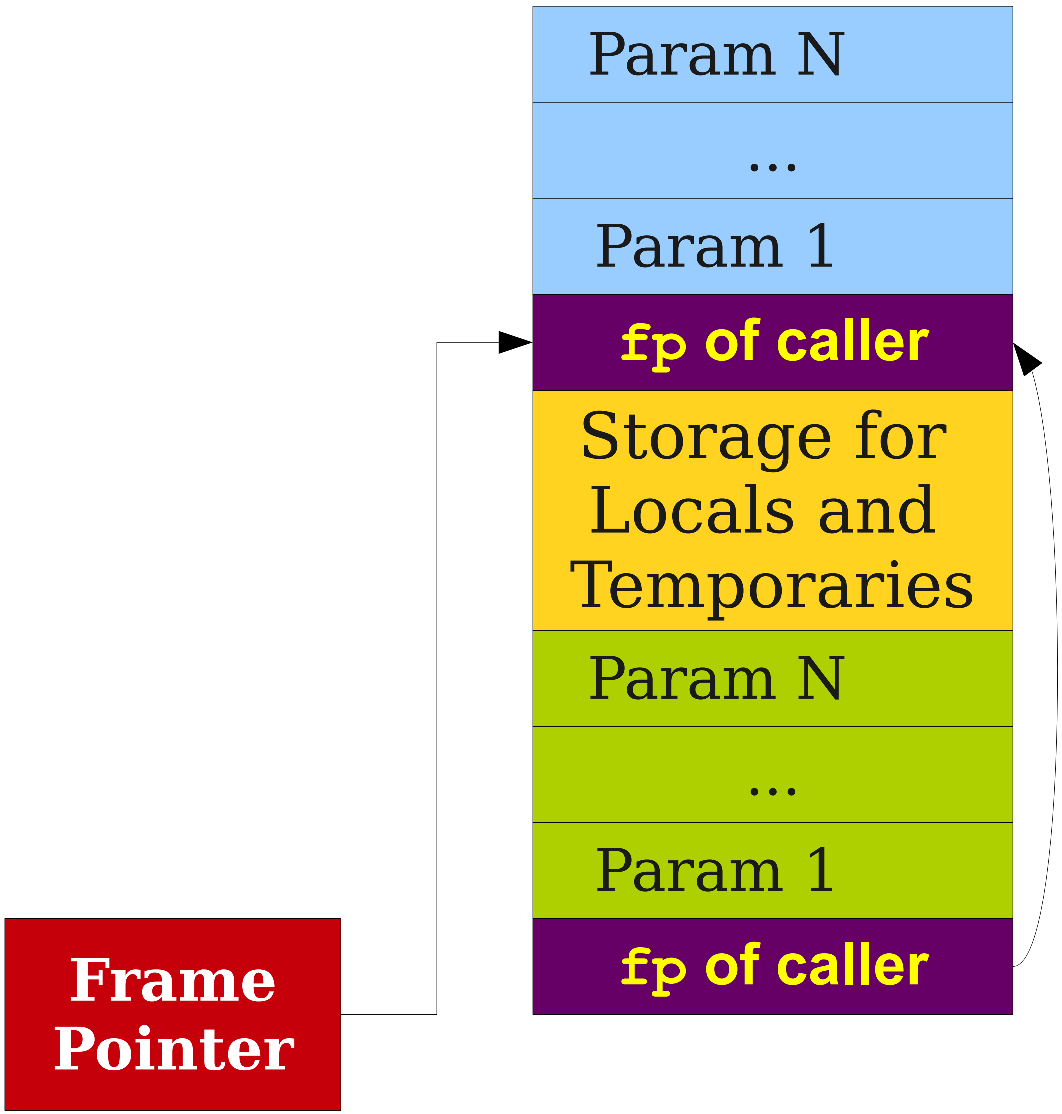
(Mostly) Physical Stack Frames



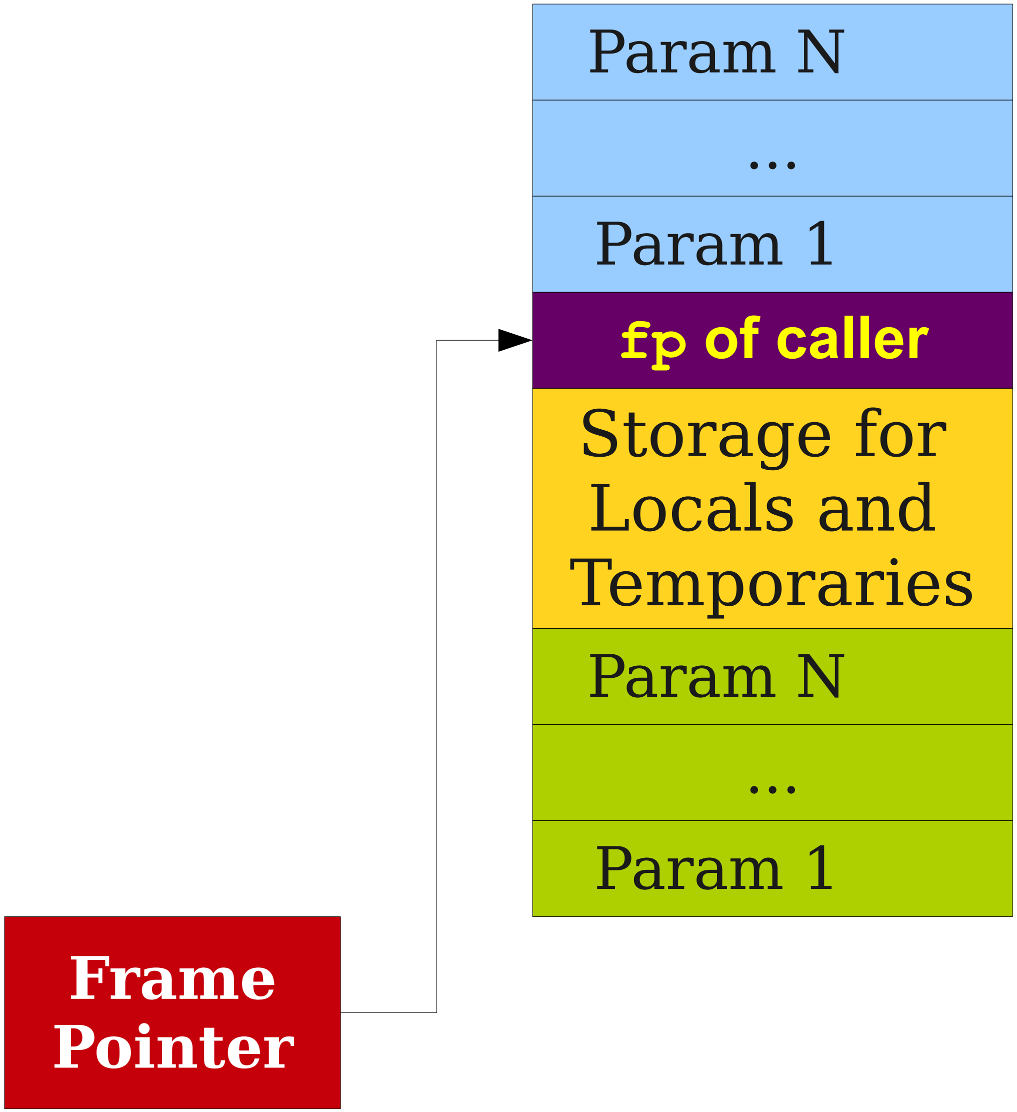
(Mostly) Physical Stack Frames



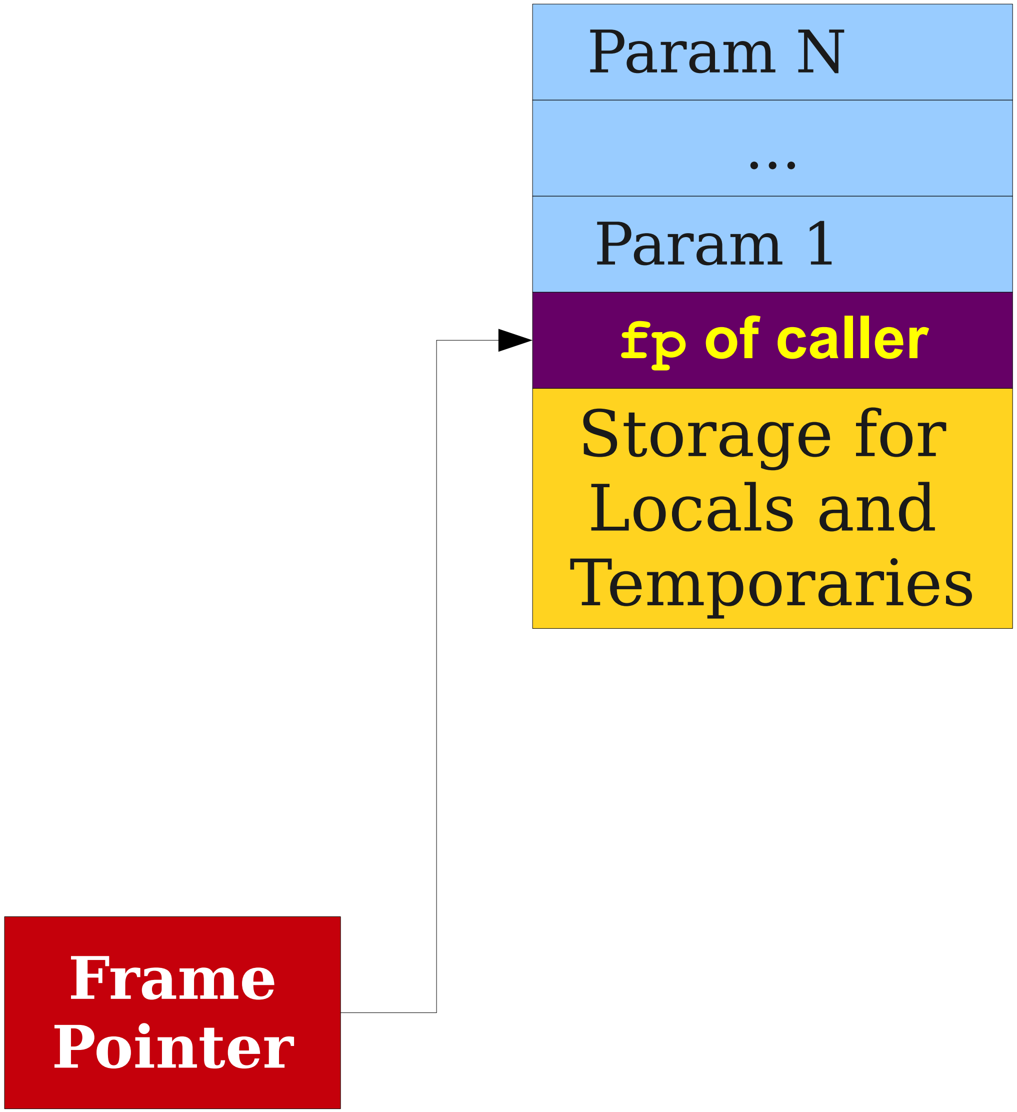
(Mostly) Physical Stack Frames



(Mostly) Physical Stack Frames



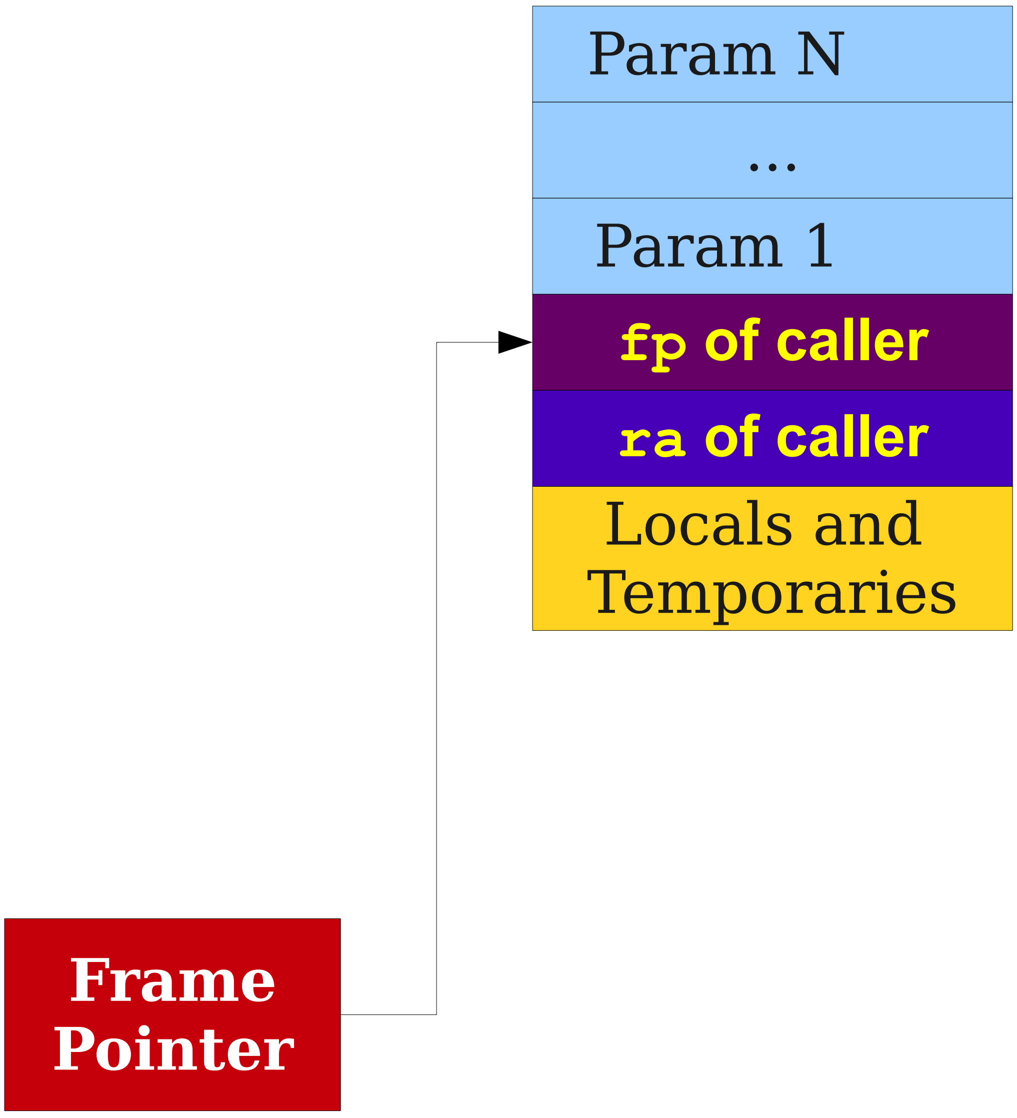
(Mostly) Physical Stack Frames



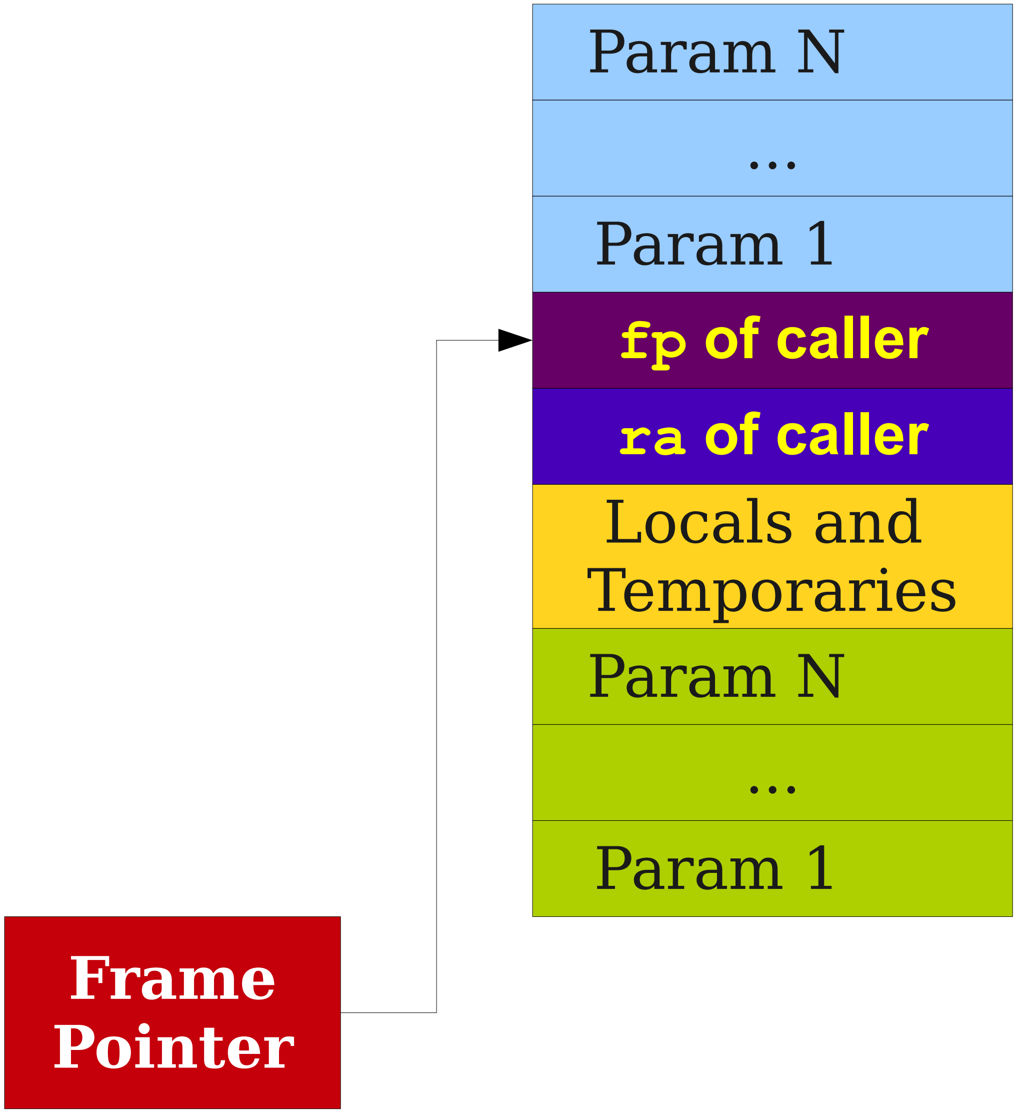
The Stored Return Address

- Internally, the processor has a special register called the **program counter** (PC) that stores the address of the next instruction to execute.
- Whenever a function returns, it needs to restore the PC so that the calling function resumes execution where it left off.
- The address of where to return is stored in MIPS in a special register called **`ra`** ("return address.")
- To allow MIPS functions to call one another, each function needs to store the previous value of **`ra`** somewhere.

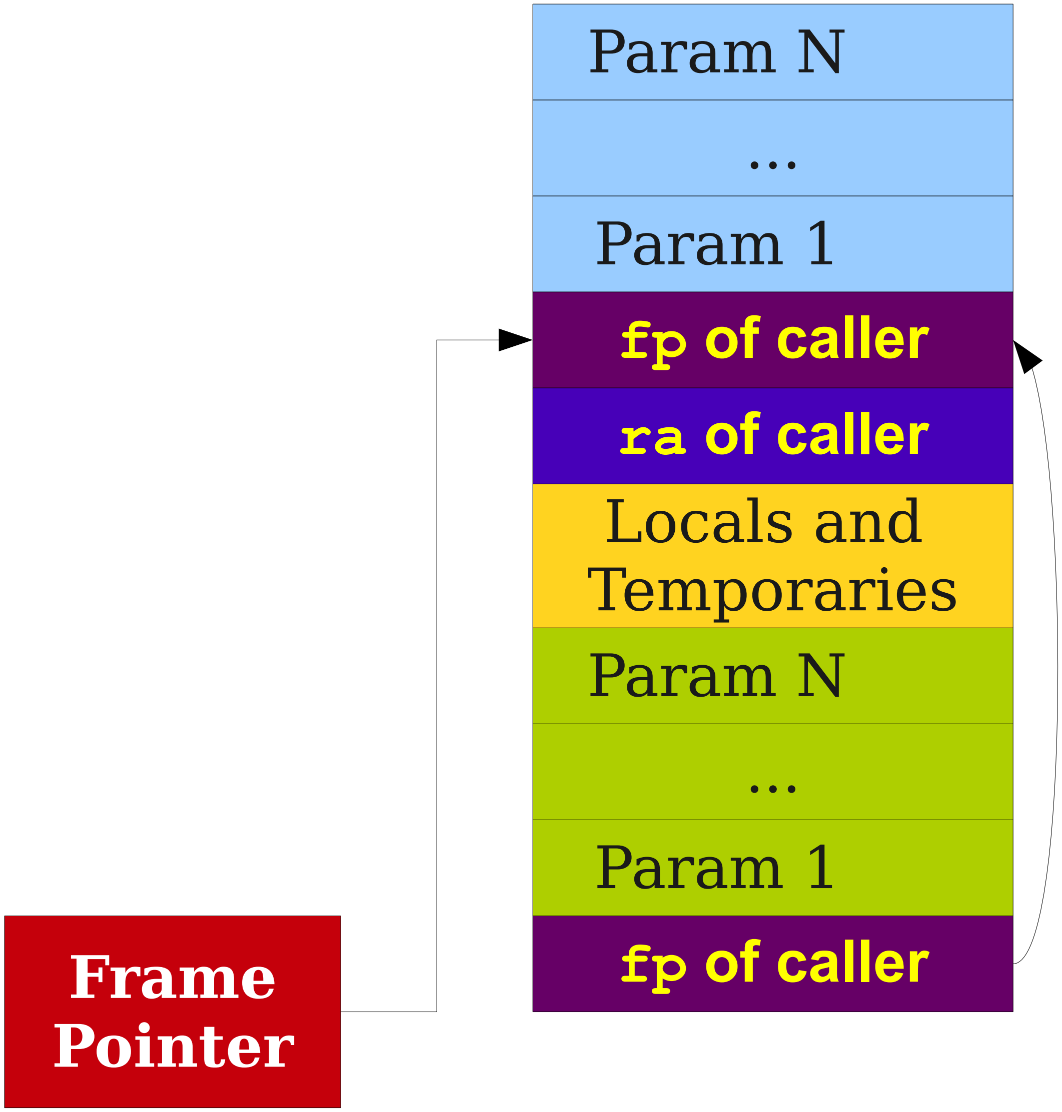
Physical Stack Frames



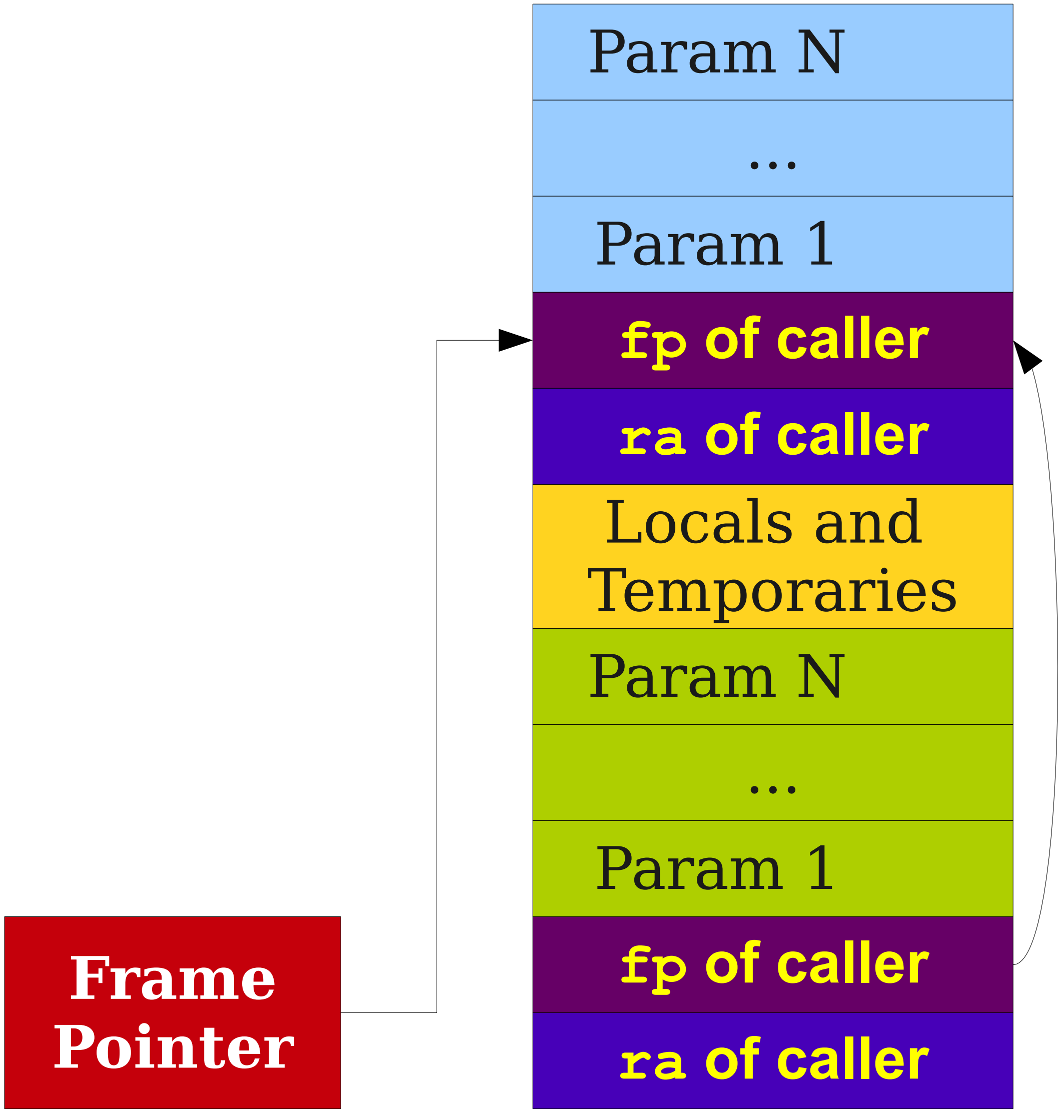
Physical Stack Frames



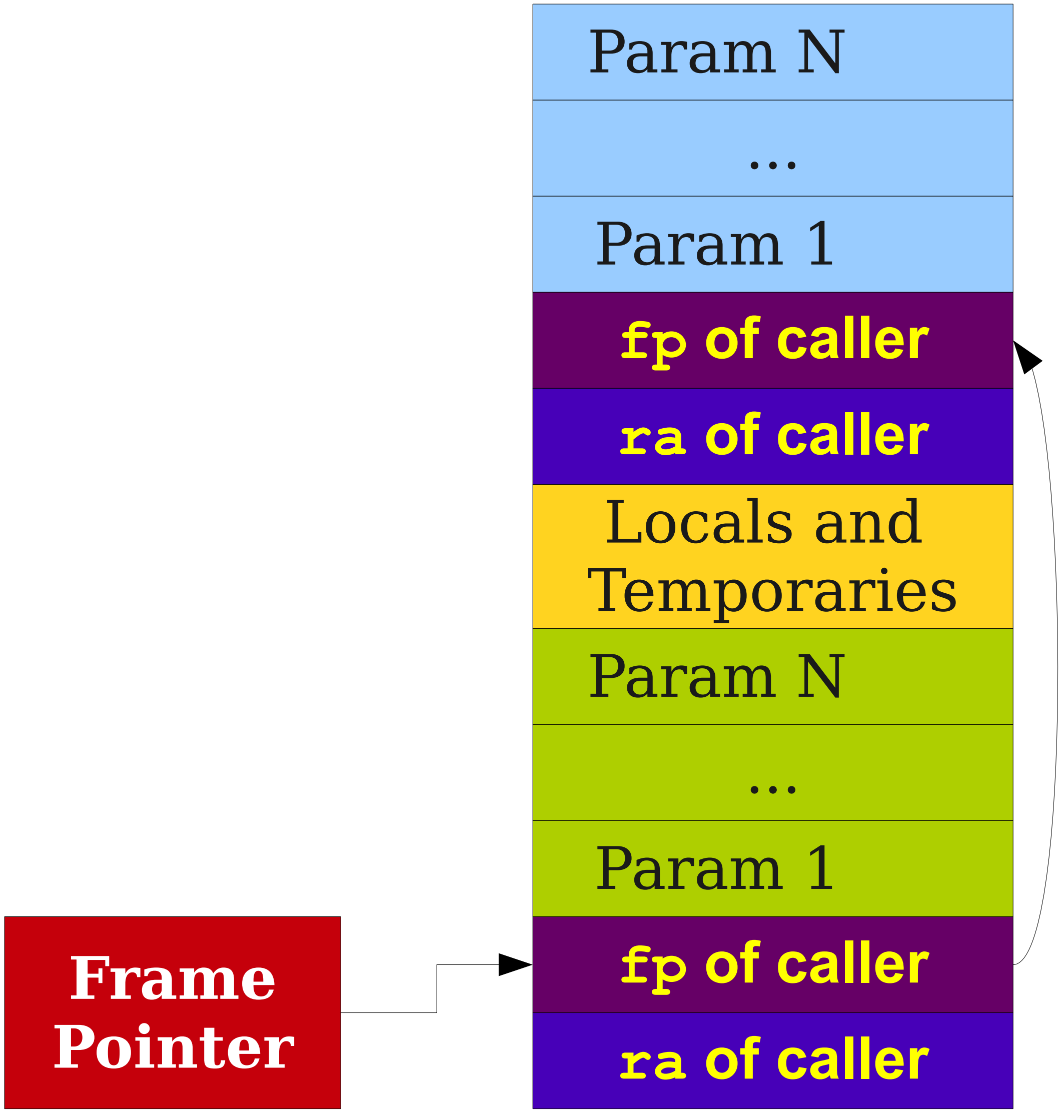
Physical Stack Frames



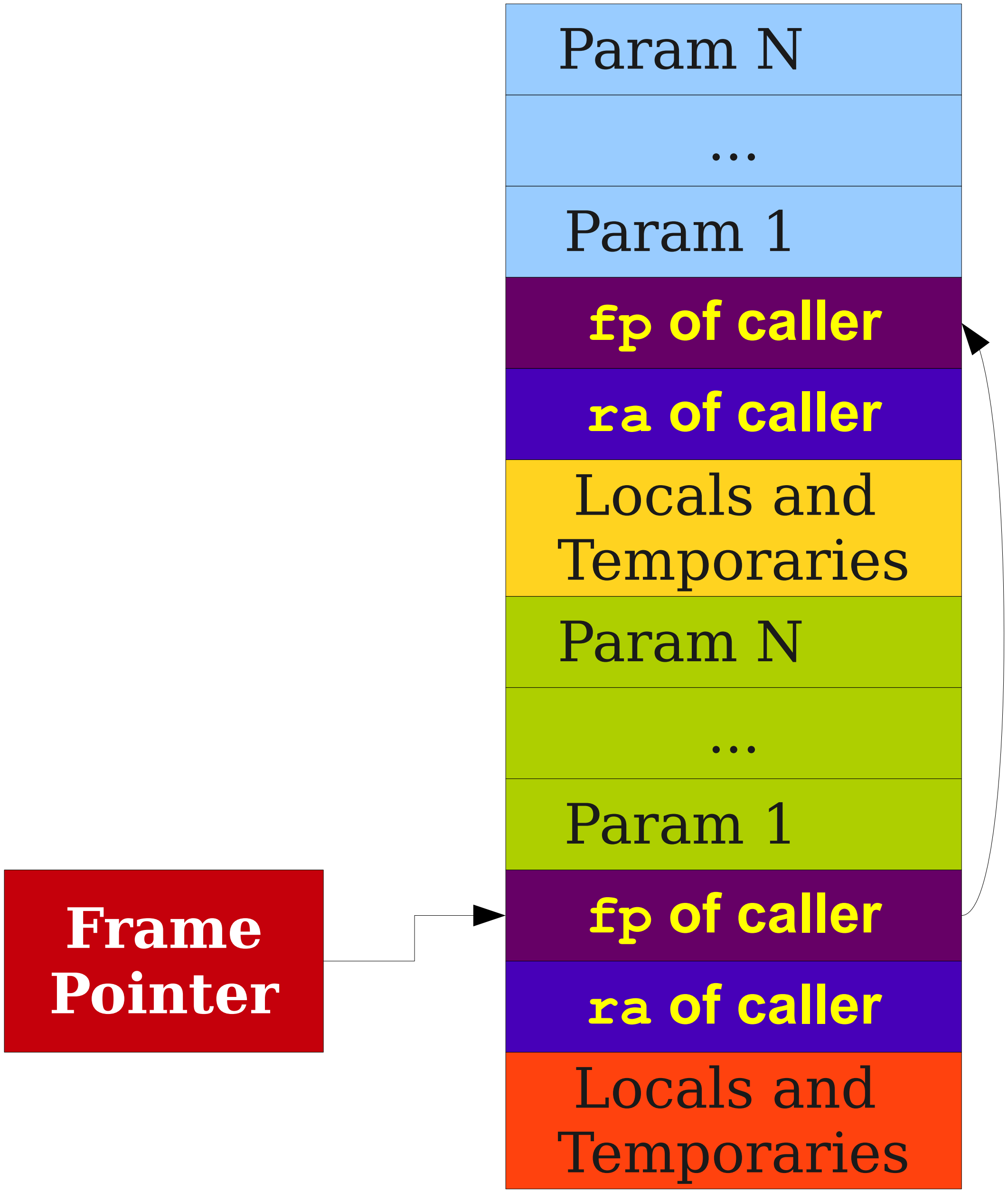
Physical Stack Frames



Physical Stack Frames

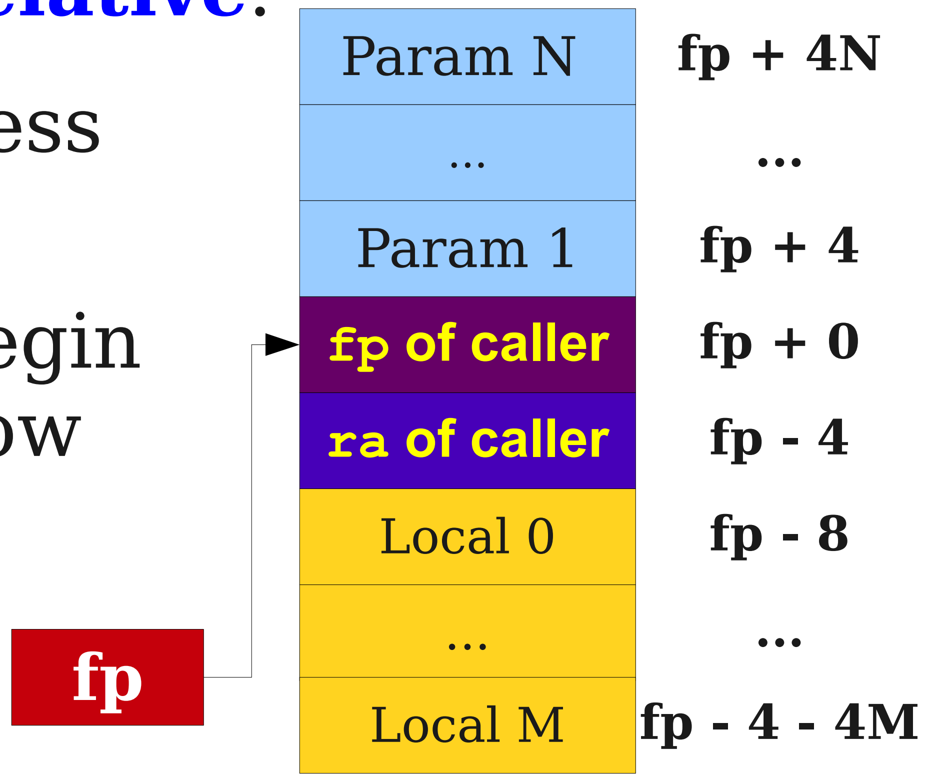


Physical Stack Frames



So What?

- In your code generator, you must assign each local variable, parameter, and temporary variable its own location.
- These locations occur in a particular stack frame and are called **fp-relative**.
- Parameters begin at address **`fp + 4`** and grow upward.
- Locals and temporaries begin at address **`fp - 8`** and grow downward



From Your Perspective

```
Location* location =
    new Location(fpRelative, +4, locName);
```

From Your Perspective

```
Location* location =
    new Location(fpRelative, +4, locName);
```

From Your Perspective

```
Location* location =
    new Location(fpRelative, +4, locName);
```

What variable does
this refer to?

And One More Thing...

```
int globalVariable;

int main() {
    globalVariable = 137;
}
```

And One More Thing...

```
int globalVariable;

int main() {
    globalVariable = 137;
}
```

And One More Thing...

```
int globalVariable;

int main() {
    globalVariable = 137;
}
```

Where is this stored?

The Global Pointer

- MIPS also has a register called the **global pointer** (**gp**) that points to globally accessible storage.
- Memory pointed at by the global pointer is treated as an array of values that grows upward.
- You must choose an offset into this array for each global variable.

From Your Perspective

```
Location* global =
    new Location(gpRelative, +8, locName);
```

From Your Perspective

```
Location* global =
    new Location(gpRelative, +8, locName);
```

Summary of Memory Layout

- Most details abstracted away by IR format.
- Remember:
  - Parameters start at **fp + 4** and grow upward.
  - Locals start at **fp – 8** and grow downward.
  - Globals start at **gp + 0** and grow upward.
- You will need to write code to assign variables to these locations.

TAC for Objects, Part I

```
class A {
    void fn(int x) {
        int y;
        y = x;
    }
}

int main() {
    A a;
    a.fn(137);
}
```

TAC for Objects, Part I

```
class A {
    void fn(int x) {
        int y;
        y = x;
    }
}

int main() {
    A a;
    a.fn(137);
}
```

```
_A.fn:
    BeginFunc 4;
    y = x;
    EndFunc;

main:
    BeginFunc 8;
    _t0 = 137;
    PushParam _t0;
    PushParam a;
    LCall _A.fn;
    PopParams 8;
    EndFunc;
```

TAC for Objects, Part I

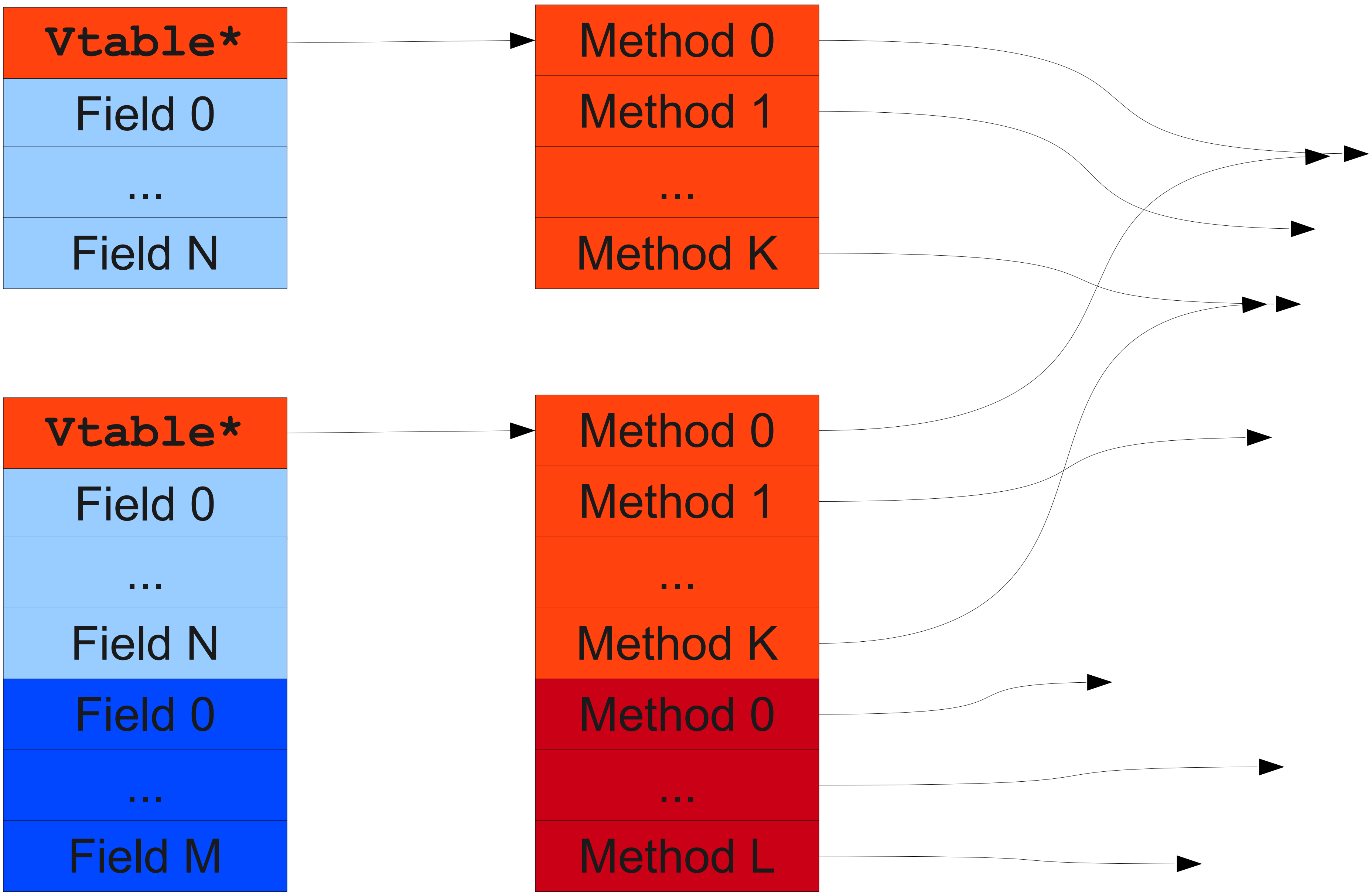
```
class A {
    void fn(int x) {
        int y;
        y = x;
    }
}

int main() {
    A a;
    a.fn(137);
}
```

```
_A.fn:
    BeginFunc 4;
    y = x;
    EndFunc;

main:
    BeginFunc 8;
    _t0 = 137;
    PushParam _t0;
    PushParam a;
    LCall _A.fn;
    PopParams 8;
    EndFunc;
```

A Reminder: Object Layout



TAC for Objects, Part II

```
class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main() {
    A a;
    a.fn(137);
}
```

TAC for Objects, Part II

```
class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main() {
    A a;
    a.fn(137);
}
```

```
_A.fn:
    BeginFunc 4;
    *(this + 4) = x;
    x = *(this + 8);
    EndFunc;

main:
    BeginFunc 8;
    _t0 = 137;
    PushParam _t0;
    PushParam a;
    LCall _A.fn;
    PopParams 8;
    EndFunc;
```

TAC for Objects, Part II

```
class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main() {
    A a;
    a.fn(137);
}
```

```
_A.fn:
    BeginFunc 4;
    *(this + 4) = x;
    x = *(this + 8);
    EndFunc;

main:
    BeginFunc 8;
    _t0 = 137;
    PushParam _t0;
    PushParam a;
    LCall _A.fn;
    PopParams 8;
    EndFunc;
```

TAC for Objects, Part II

```
class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main() {
    A a;
    a.fn(137);
}
```

```
_A.fn:
    BeginFunc 4;
    *(this + 4) = x;
    x = *(this + 8);
    EndFunc;

main:
    BeginFunc 8;
    _t0 = 137;
    PushParam _t0;
    PushParam a;
    LCall _A.fn;
    PopParams 8;
    EndFunc;
```

Memory Access in TAC

- Extend our simple assignments with memory accesses:
  - `var`$_1$ `= *var`$_2$
  - `var`$_1$ `= *(var`$_2$ `+ constant)`
  - `*var`$_1$ `= var`$_2$
  - `*(var`$_1$ `+ constant) = var`$_2$
- You will need to translate field accesses into relative memory accesses.

TAC for Objects, Part III

```
class Base {
  void hi() {
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

TAC for Objects, Part III

```
class Base {
  void hi() {
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

TAC for Objects, Part III

```
class Base {
  void hi() {
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
_Base.hi:
    BeginFunc 4;
    _t0 = "Base";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
Vtable Base = _Base.hi,
    ;

_Derived.hi:
    BeginFunc 4;
    _t0 = "Derived";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
Vtable Derived = _Derived.hi,
    ;
```

TAC for Objects, Part III

```
class Base {
  void hi() {
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
_Base.hi:
    BeginFunc 4;
    _t0 = "Base";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
Vtable Base = _Base.hi,
    ;

_Derived.hi:
    BeginFunc 4;
    _t0 = "Derived";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
Vtable Derived = _Derived.hi,
    ;
```

TAC for Objects, Part III

```
class Base {
  void hi() {
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
_Base.hi:
    BeginFunc 4;
    _t0 = "Base";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
Vtable Base = _Base.hi,
    ;

_Derived.hi:
    BeginFunc 4;
    _t0 = "Derived";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
Vtable Derived = _Derived.hi,
    ;
```

TAC for Objects, Part III

```
class Base {
  void hi() {
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

TAC for Objects, Part III

```
class Base {
  void hi() {
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

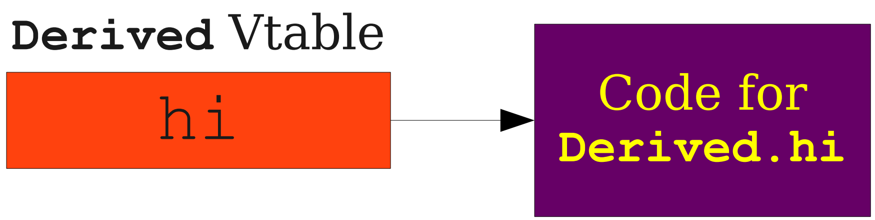
What's going on here?

Dissecting TAC

```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

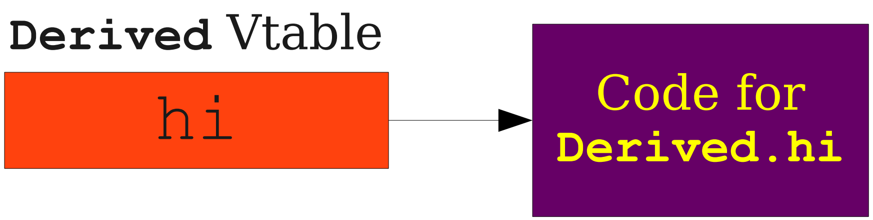
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

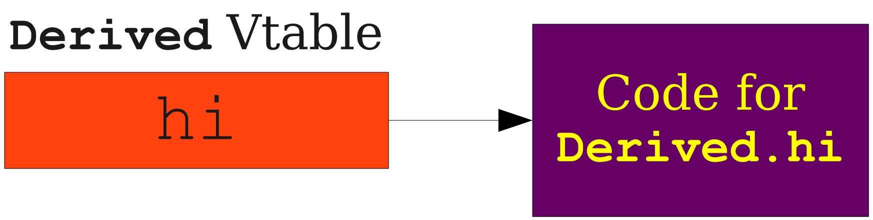
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

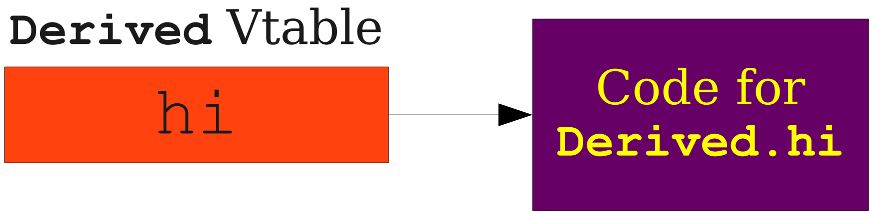
Dissecting TAC

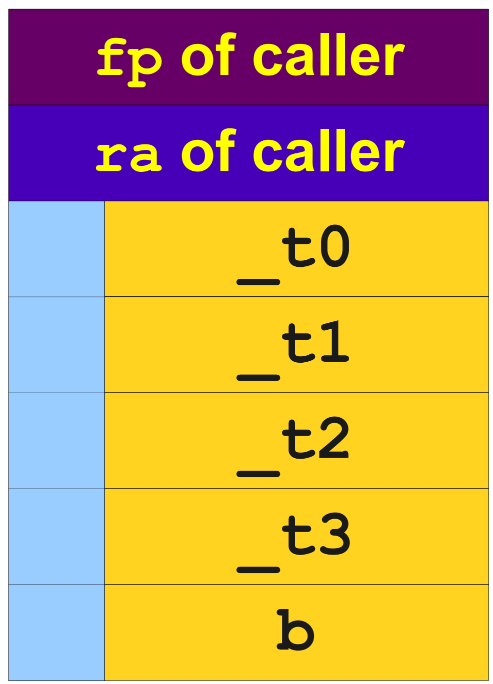



```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC

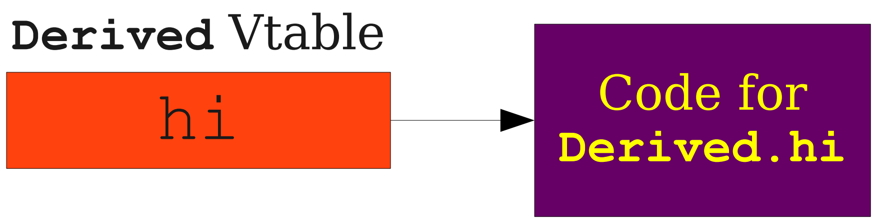


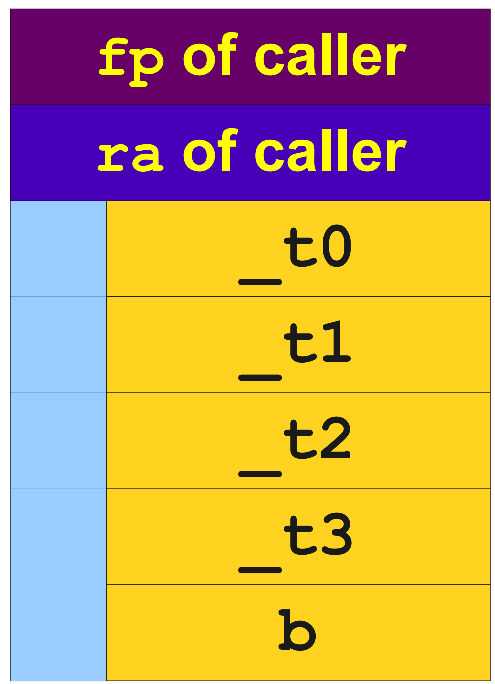



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC

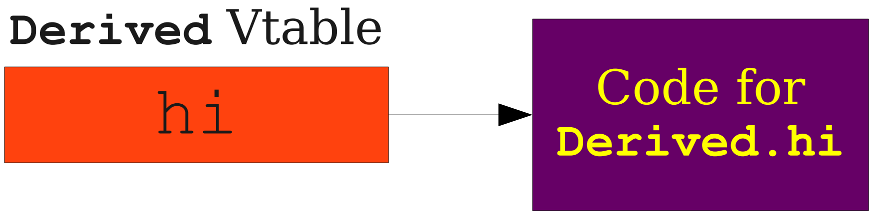


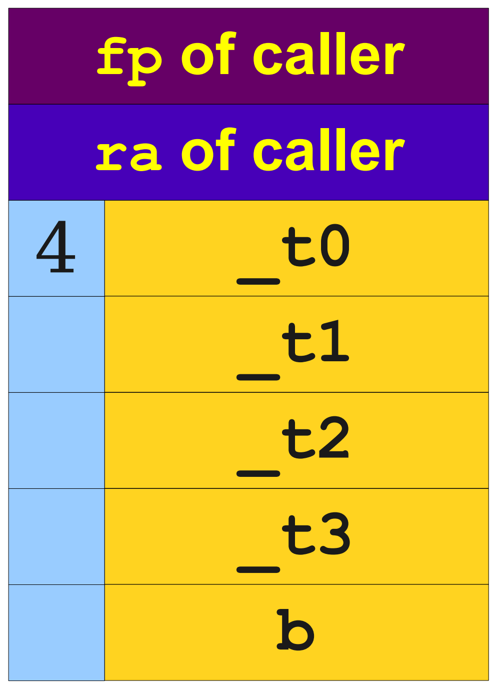



```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC

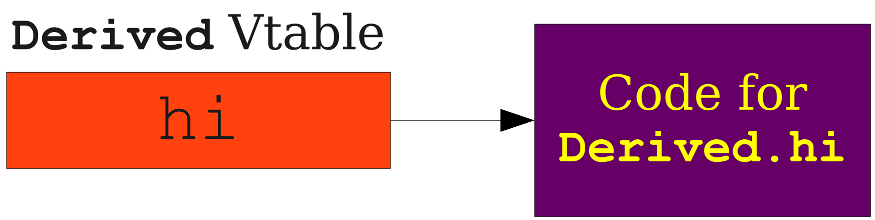


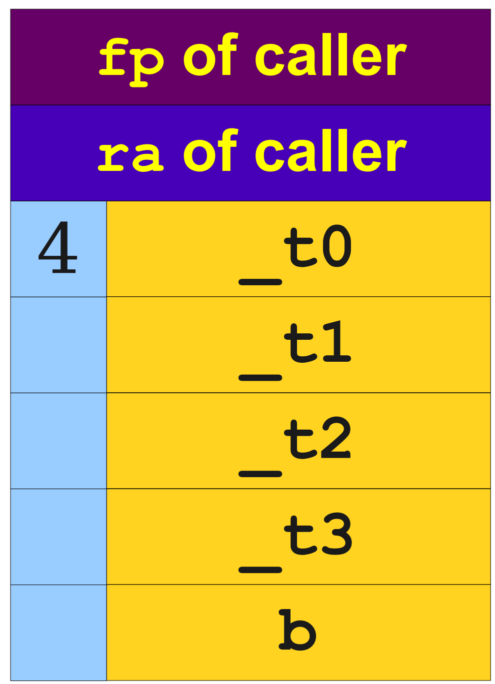



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC

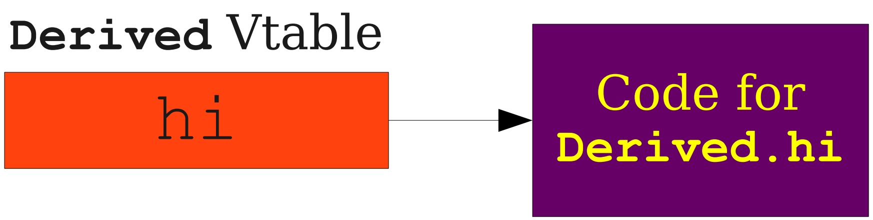


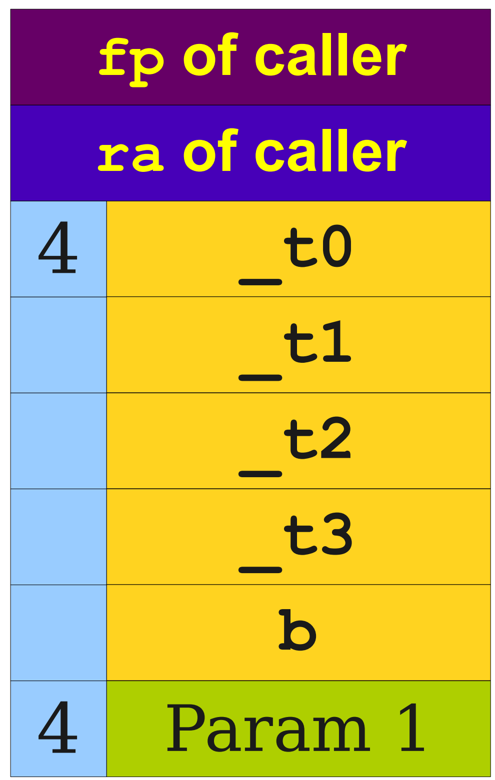



```
int main() {
      Base b;
      b = new Derived;
      b.hi();
}
```

```
main:
      BeginFunc 20;
      _t0 = 4;
      PushParam _t0;
      b = LCall _Alloc;
      PopParams 4;
      _t1 = Derived;
      *b = _t1;
      _t2 = *b;
      _t3 = *_t2;
      PushParam b;
      ACall _t3;
      PopParams 4;
      EndFunc;
```

Dissecting TAC

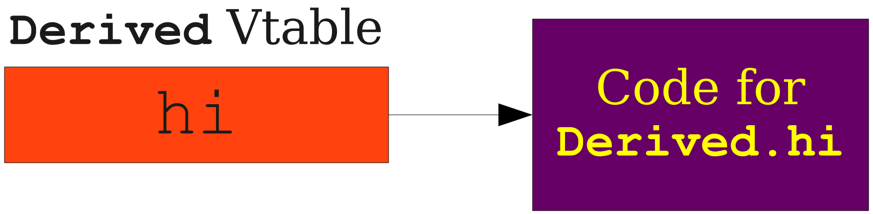


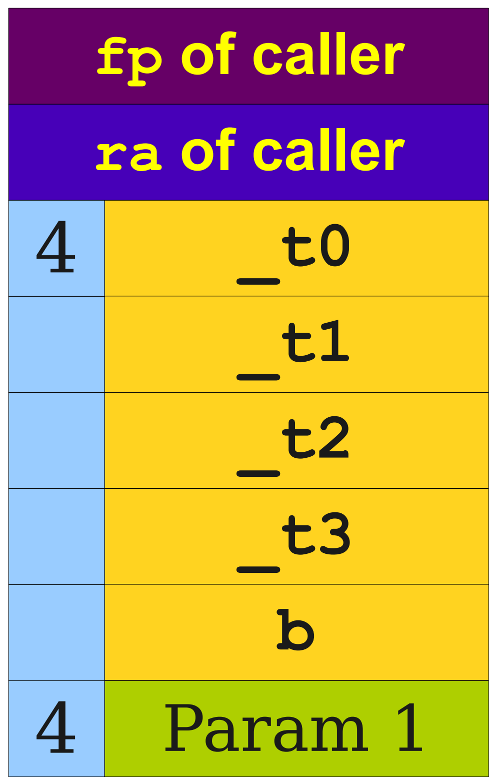



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC

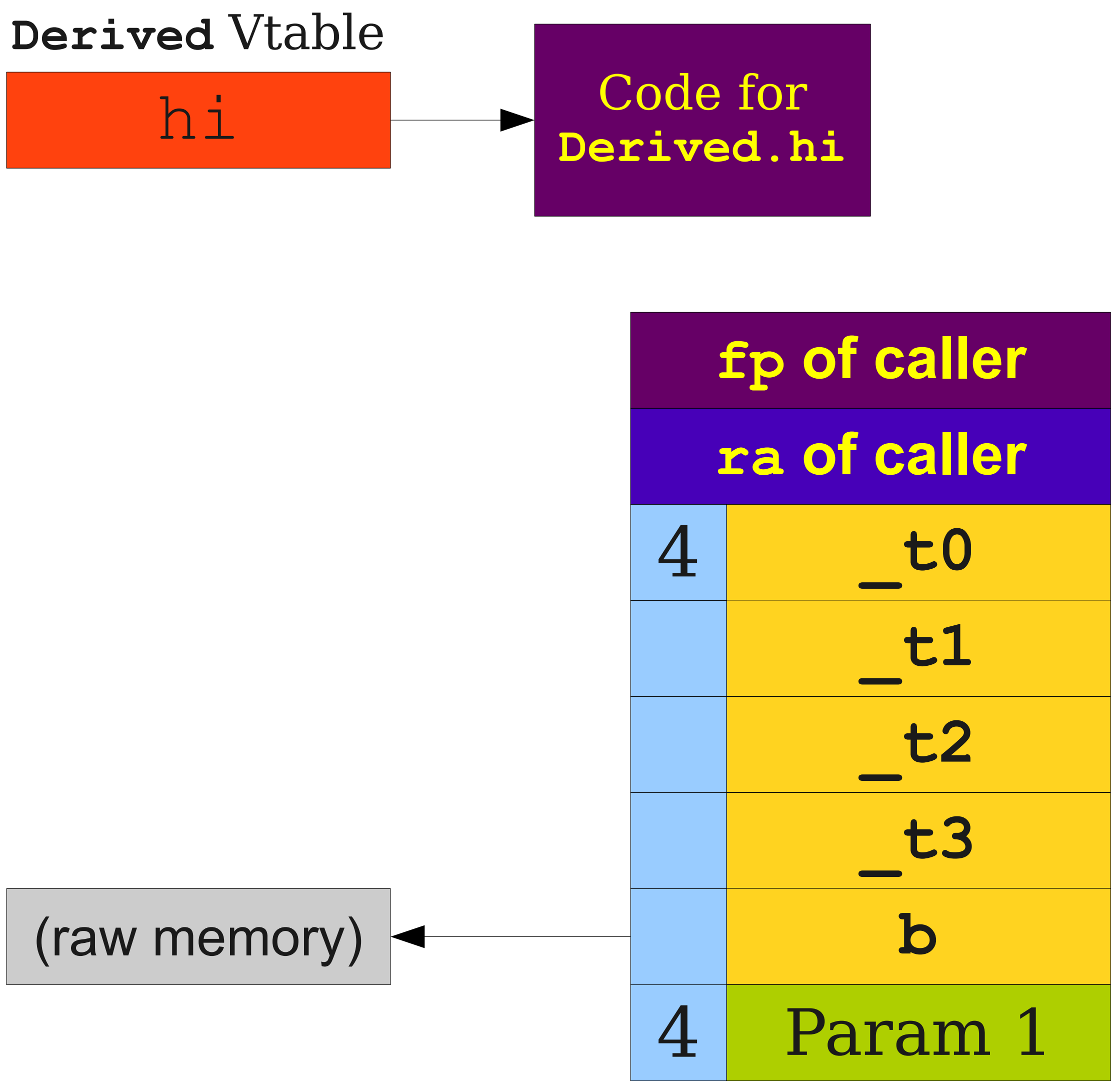





```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

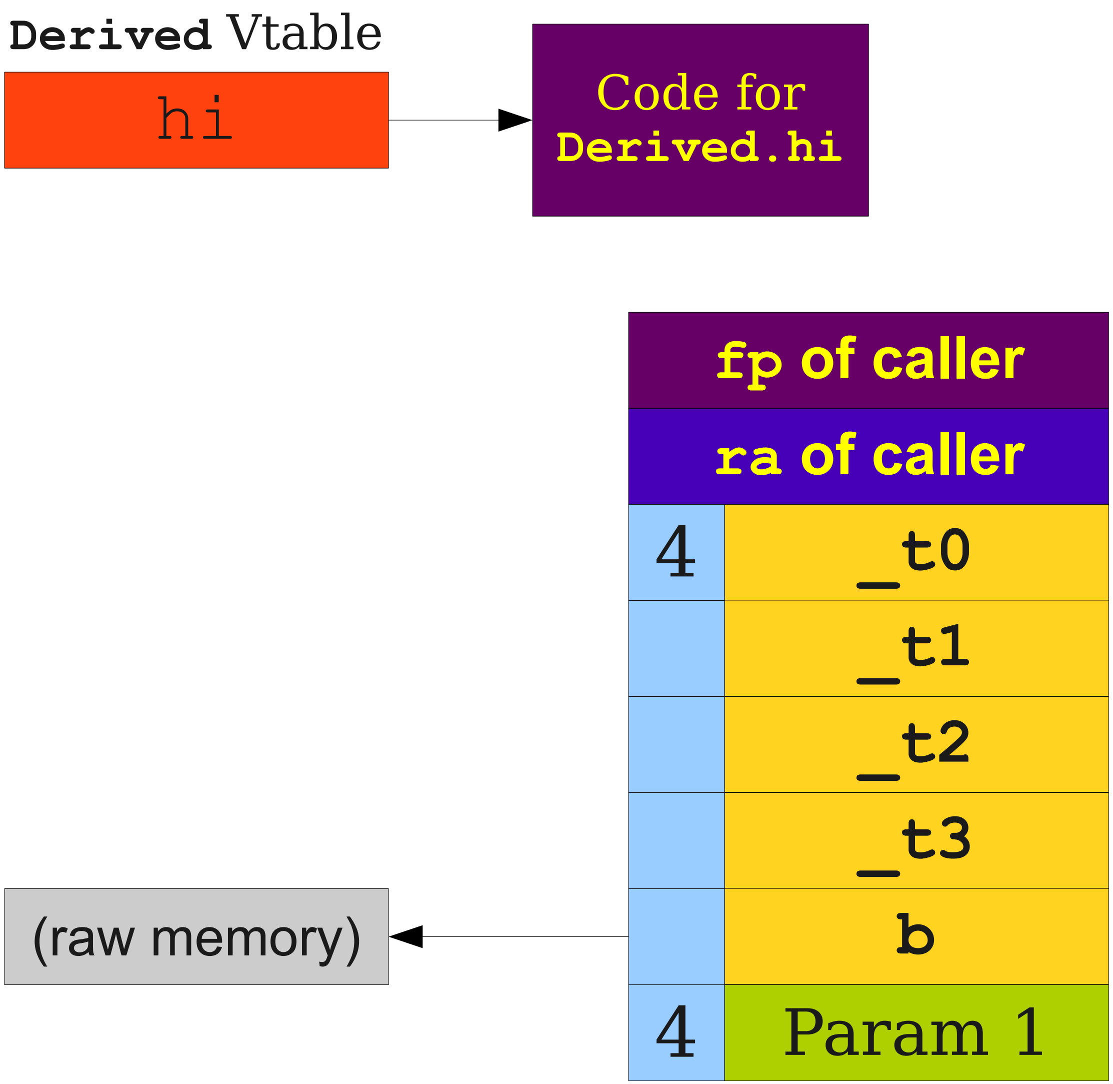
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

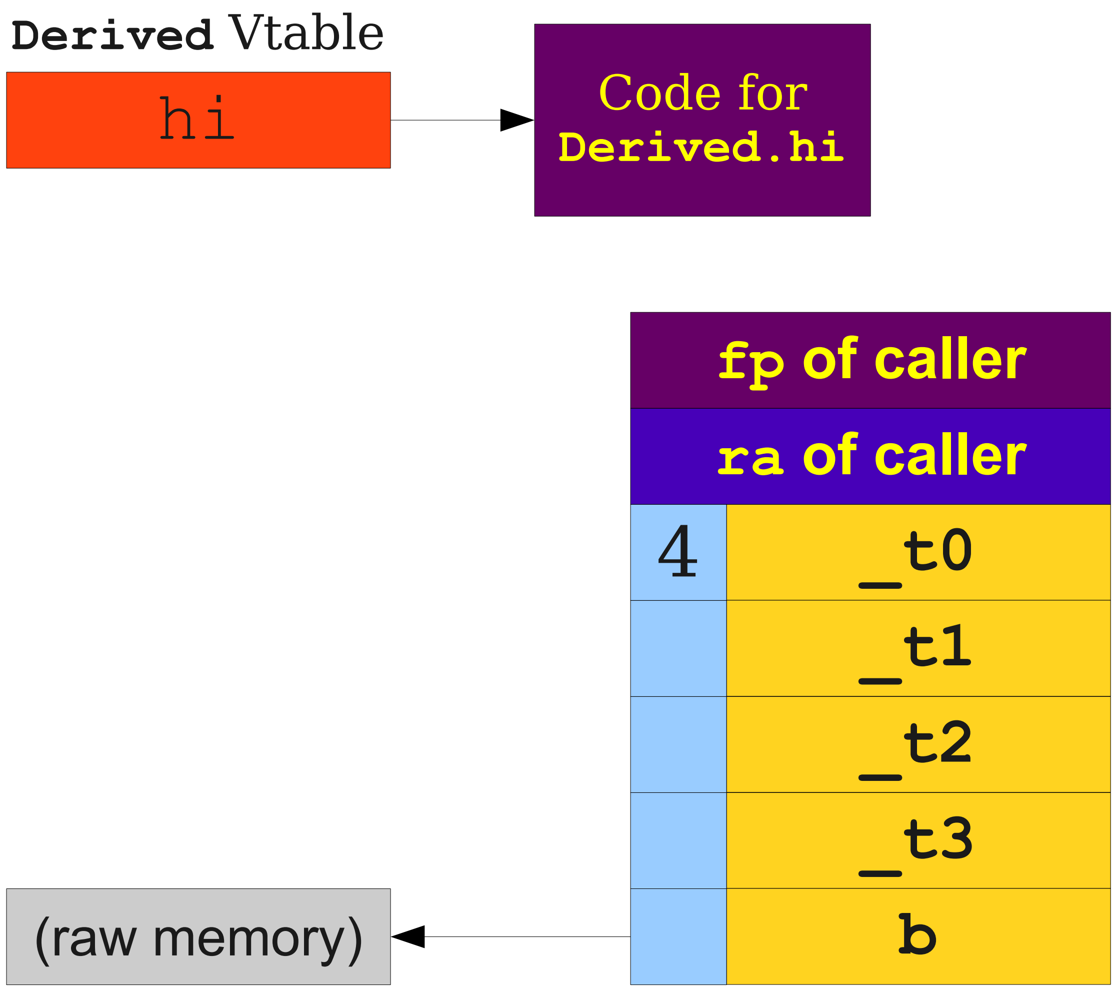
Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

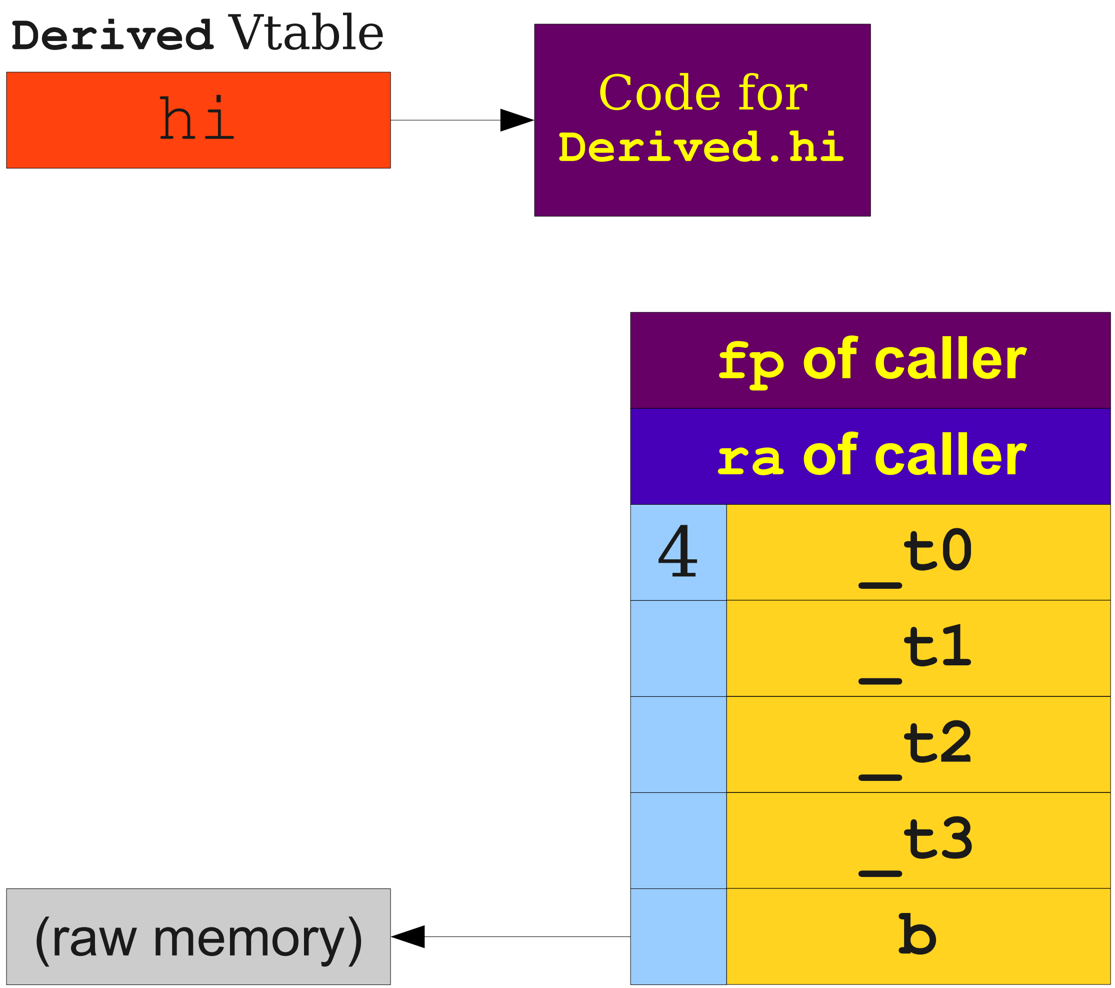
Dissecting TAC




```
int main() {
      Base b;
      b = new Derived;
      b.hi();
}
```

```
main:
      BeginFunc 20;
      _t0 = 4;
      PushParam _t0;
      b = LCall _Alloc;
      PopParams 4;
      _t1 = Derived;
      *b = _t1;
      _t2 = *b;
      _t3 = *_t2;
      PushParam b;
      ACall _t3;
      PopParams 4;
      EndFunc;
```

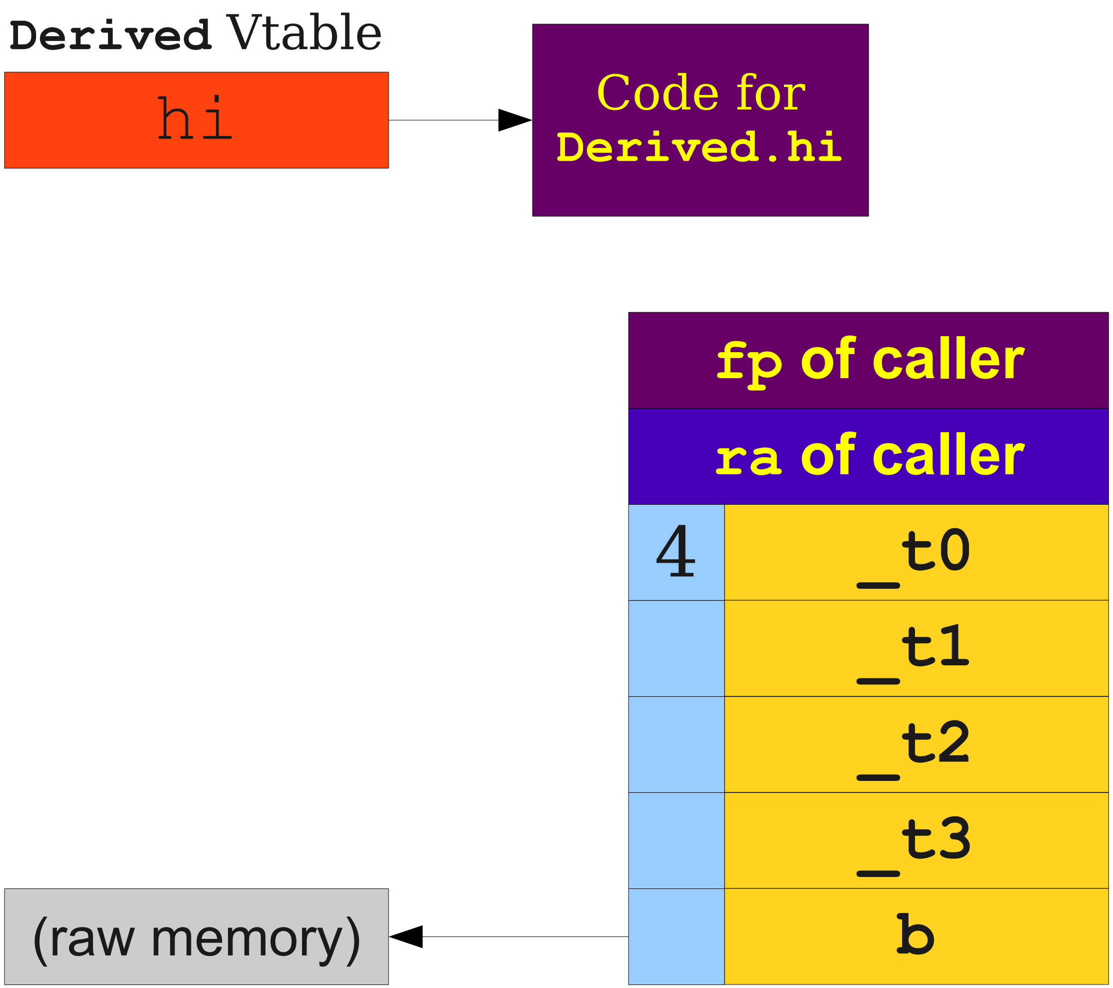
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC

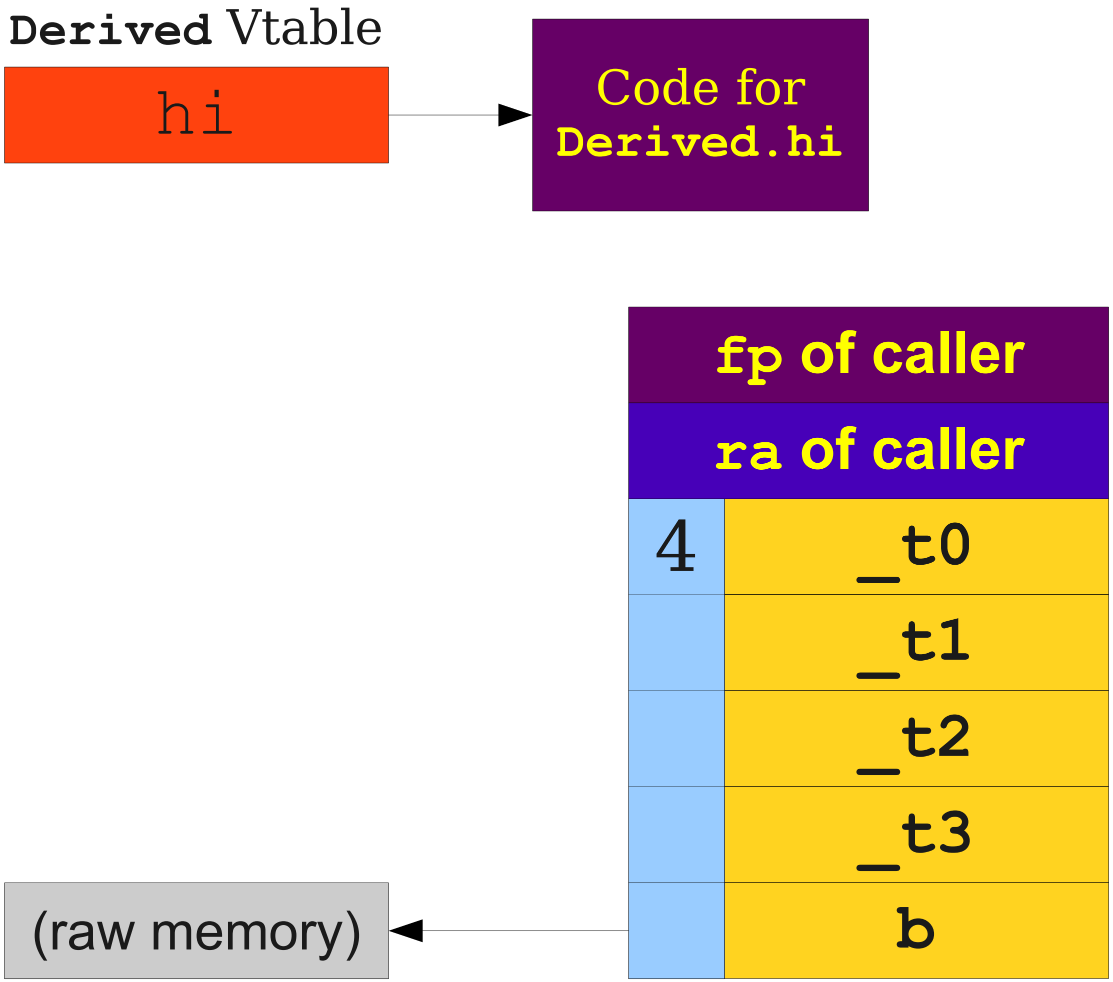



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

Allocate Object

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC

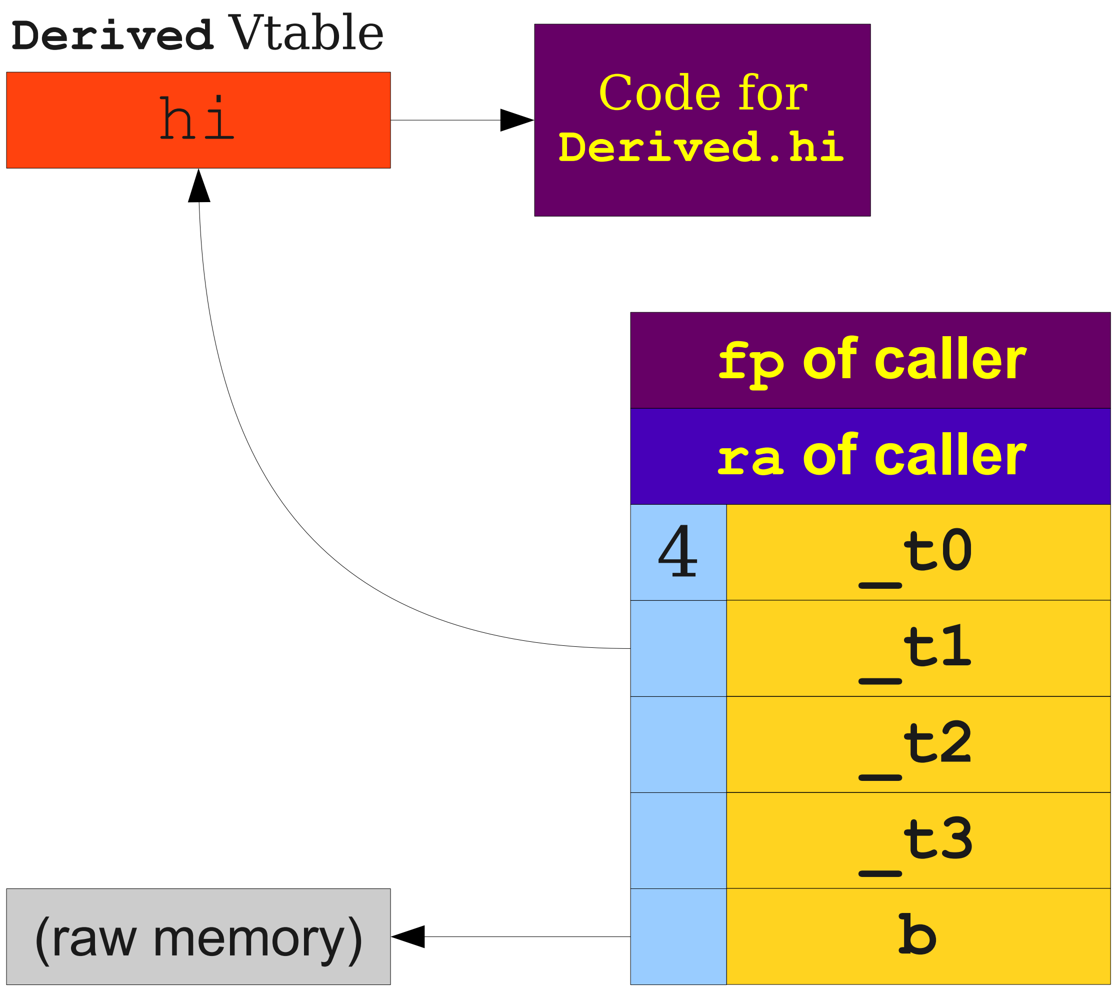



```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Allocate Object

Dissecting TAC

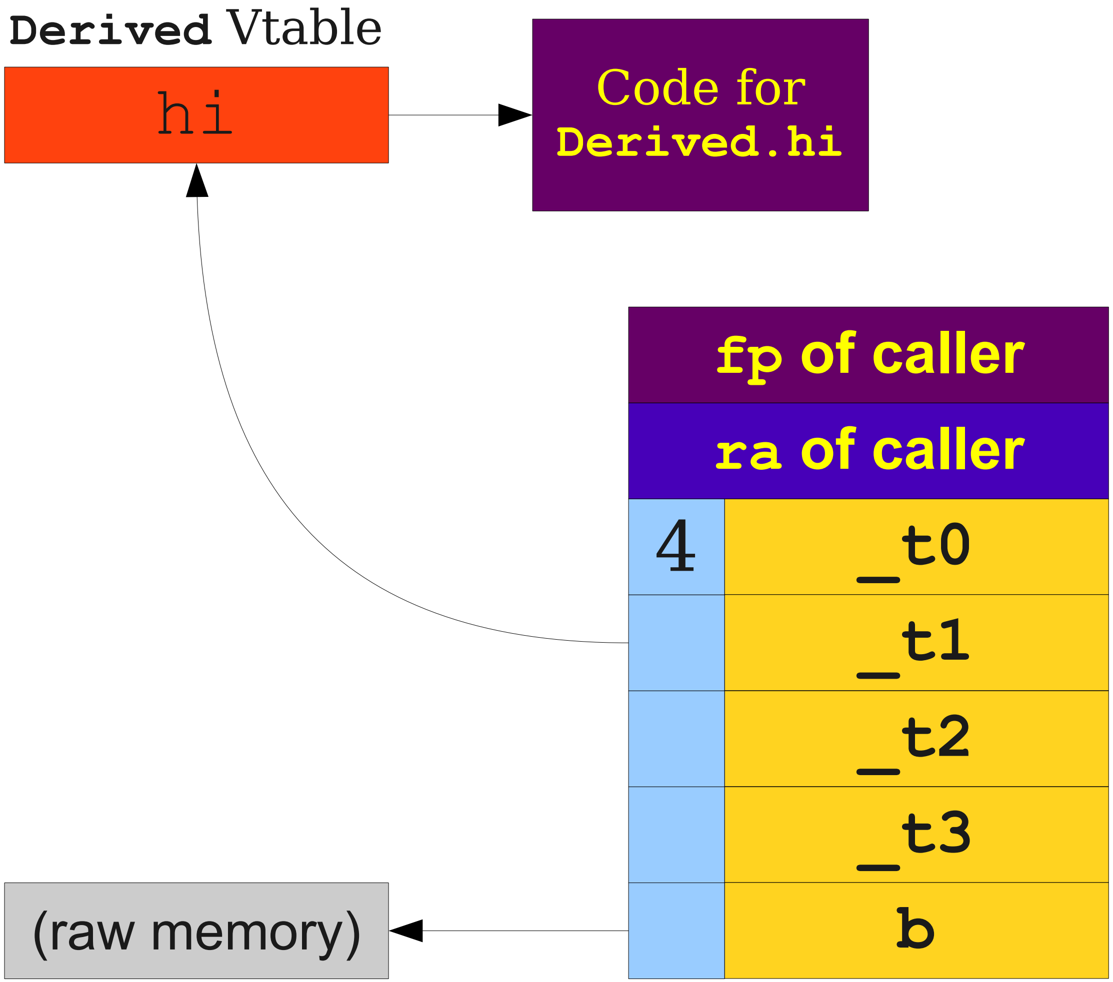



```
int main() {
      Base b;
      b = new Derived;
      b.hi();
}
```

```
main:
      BeginFunc 20;
      _t0 = 4;
      PushParam _t0;
      b = LCall _Alloc;
      PopParams 4;
      _t1 = Derived;
      *b = _t1;
      _t2 = *b;
      _t3 = *_t2;
      PushParam b;
      ACall _t3;
      PopParams 4;
      EndFunc;
```

Allocate Object

Dissecting TAC

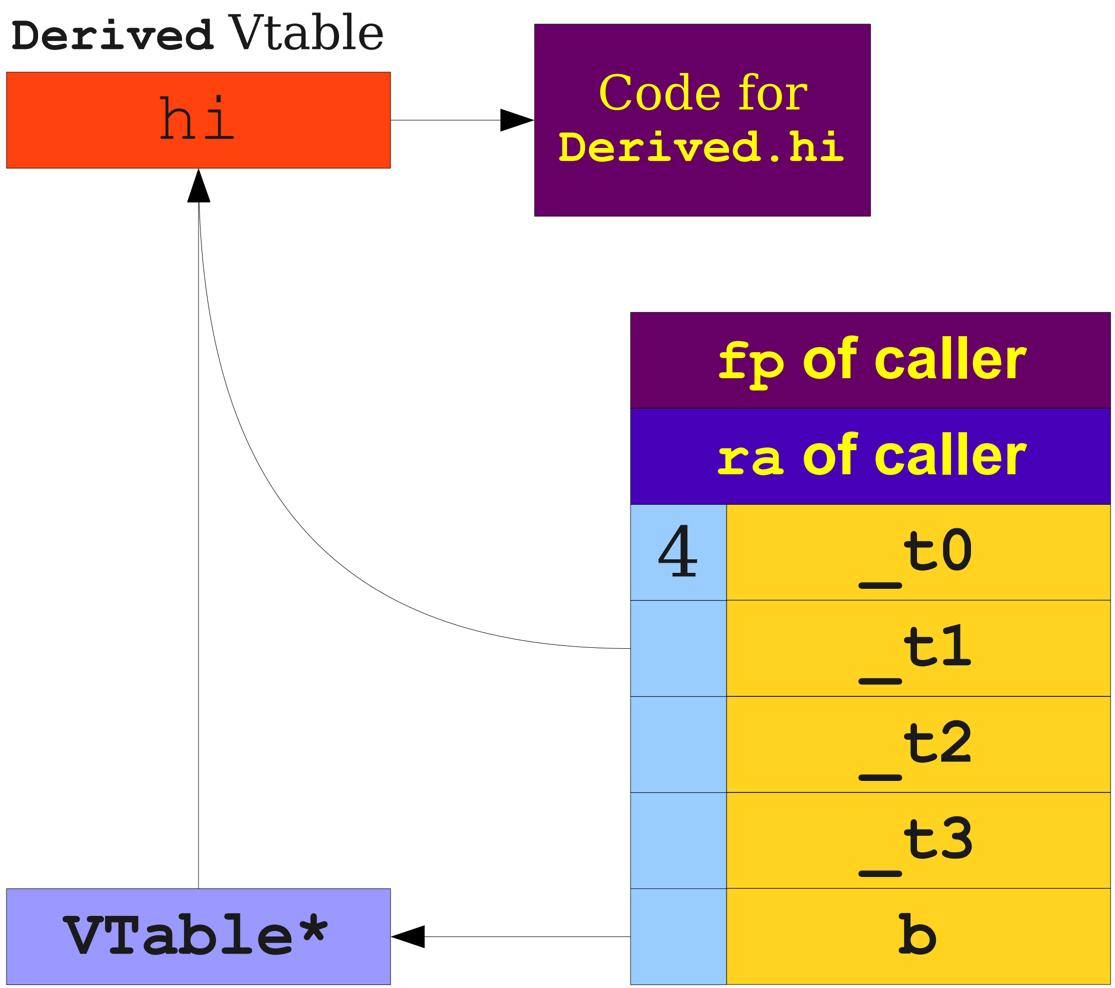



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Dissecting TAC

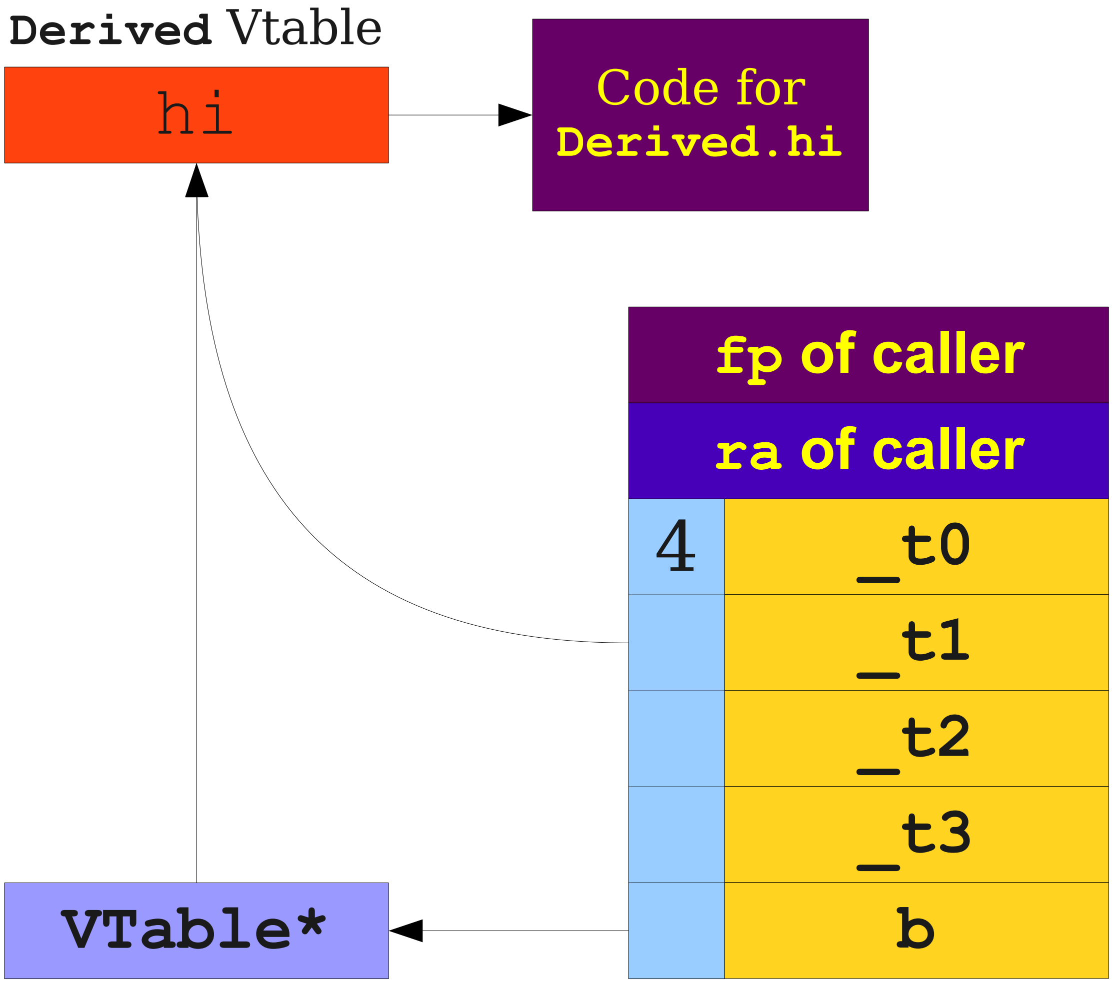



```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Allocate Object

Dissecting TAC

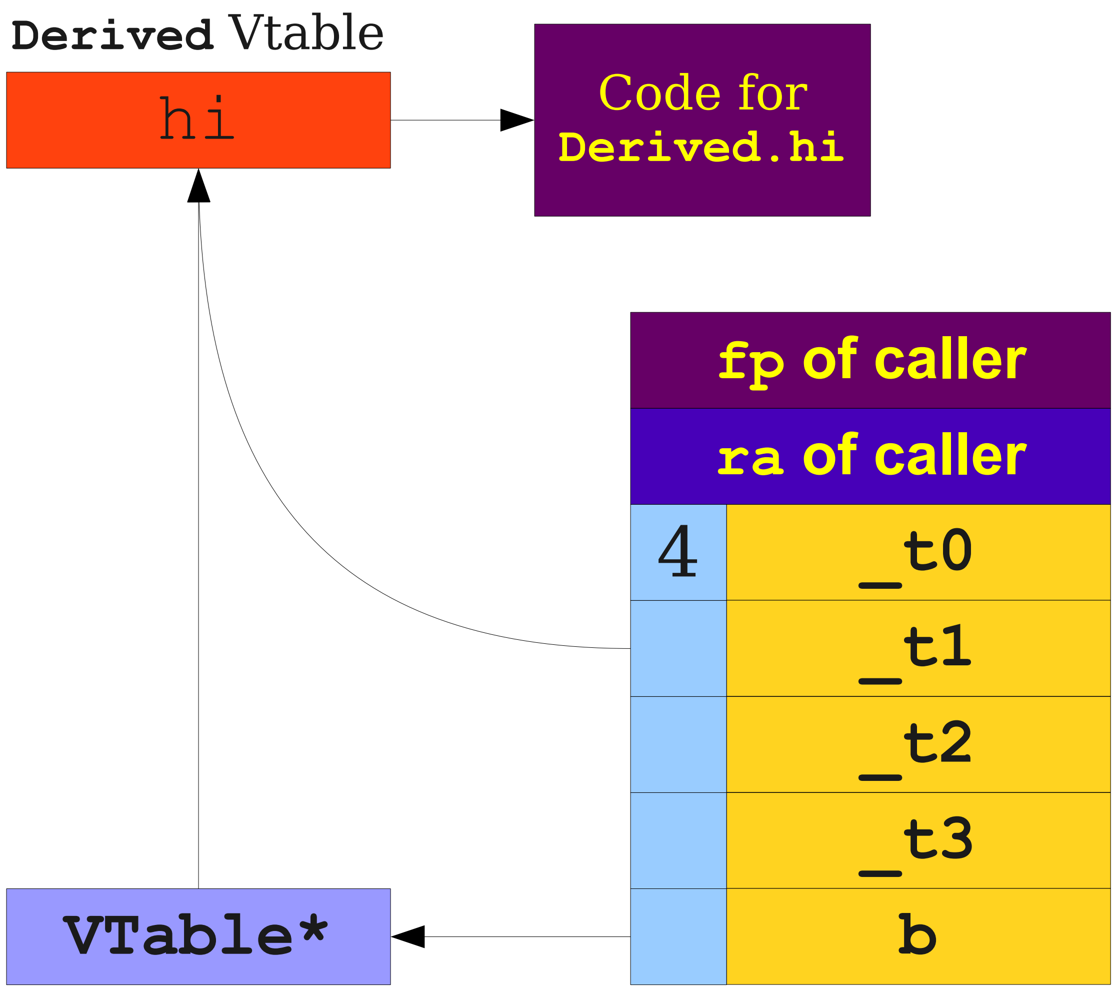



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

Allocate Object

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC



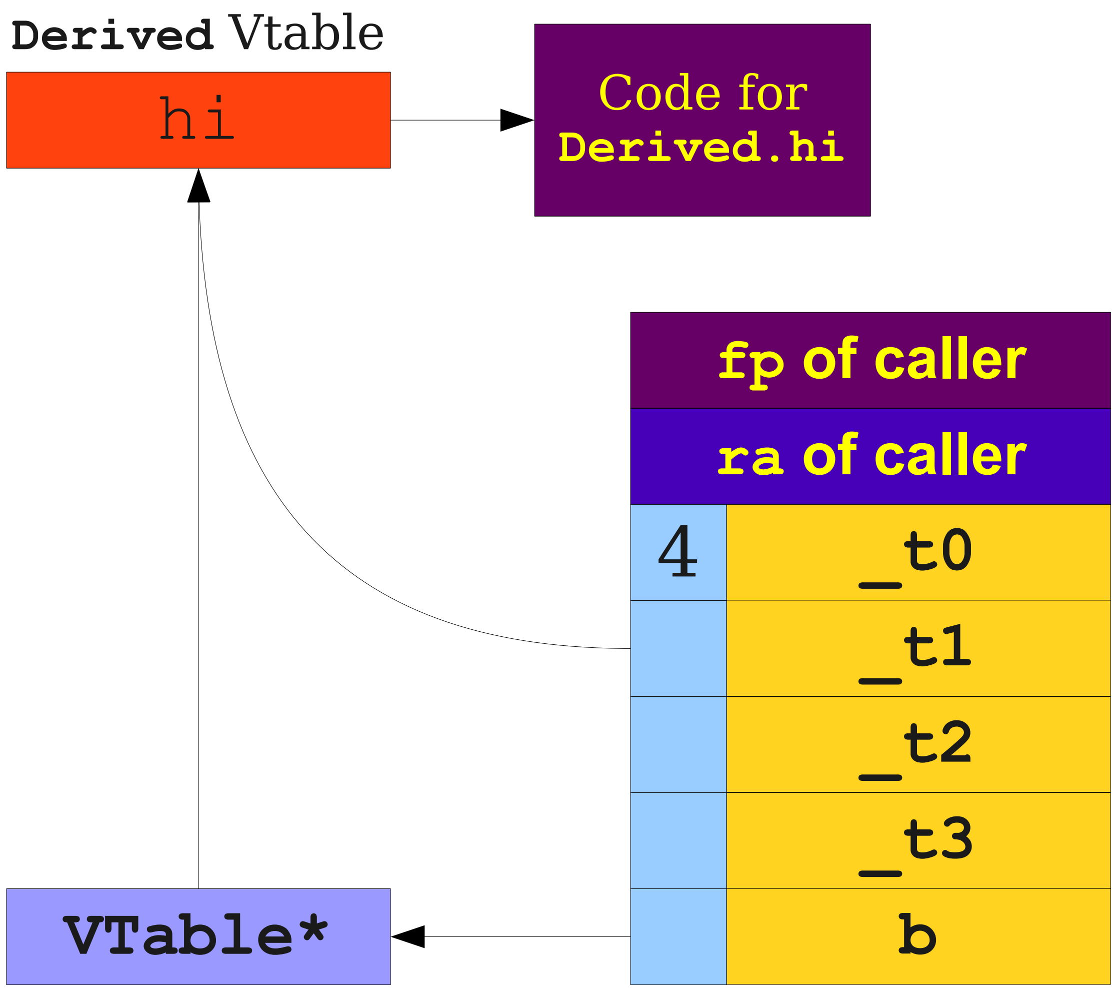

```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Set Vtable

Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
```

Allocate Object

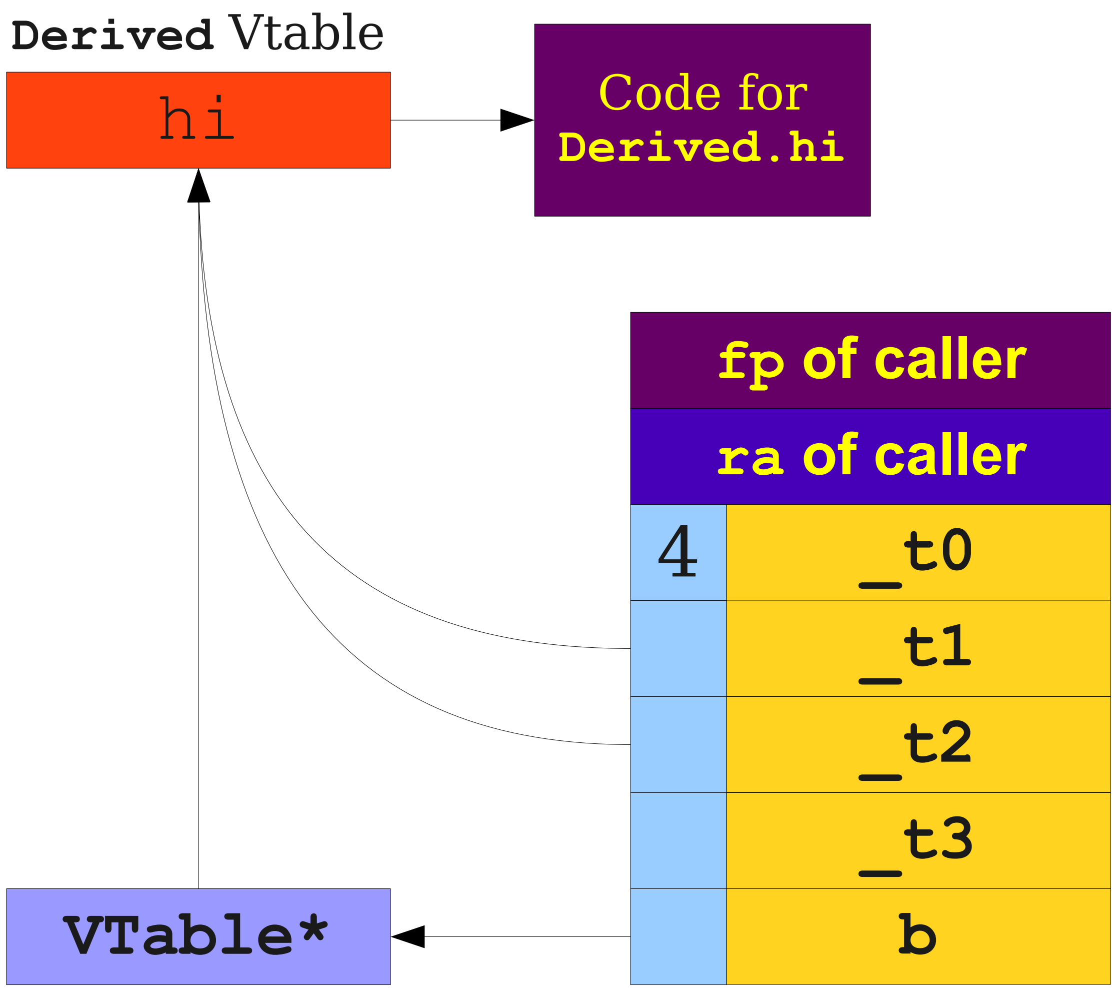
```
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
```

Set Vtable

```
     _t1 = Derived;
     *b = _t1;
```

```
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
```

Allocate Object

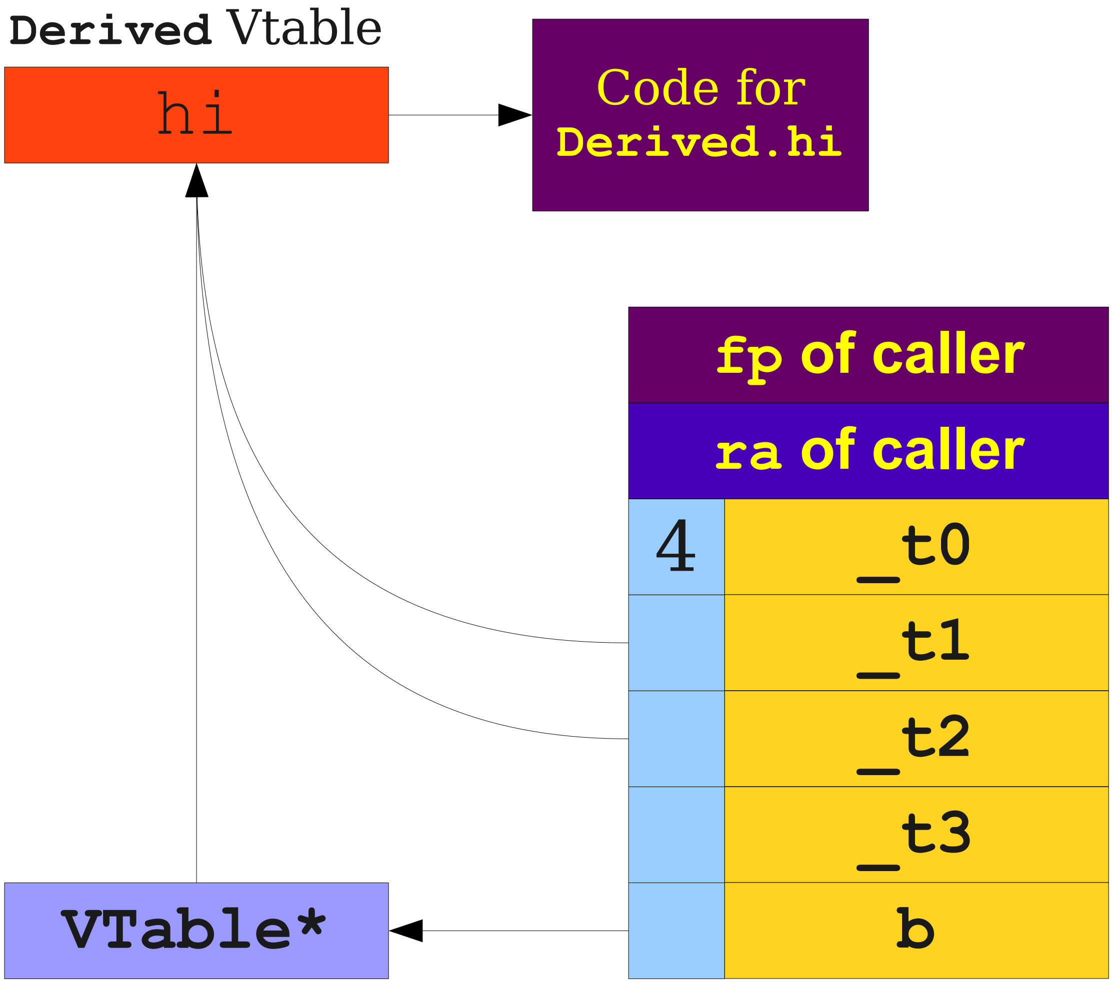
```
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
```

Set Vtable

```
     _t1 = Derived;
     *b = _t1;
```

```
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC



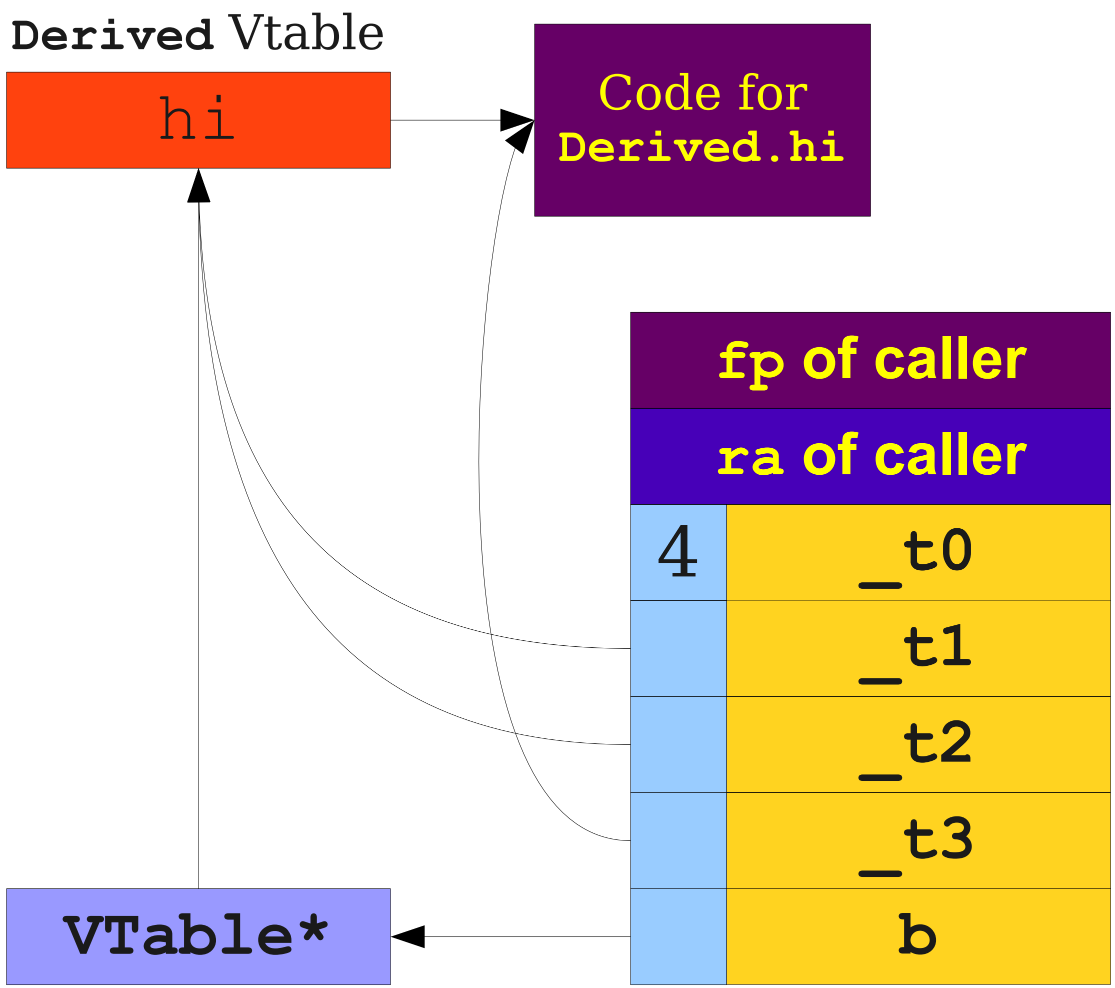

```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Set Vtable

Dissecting TAC



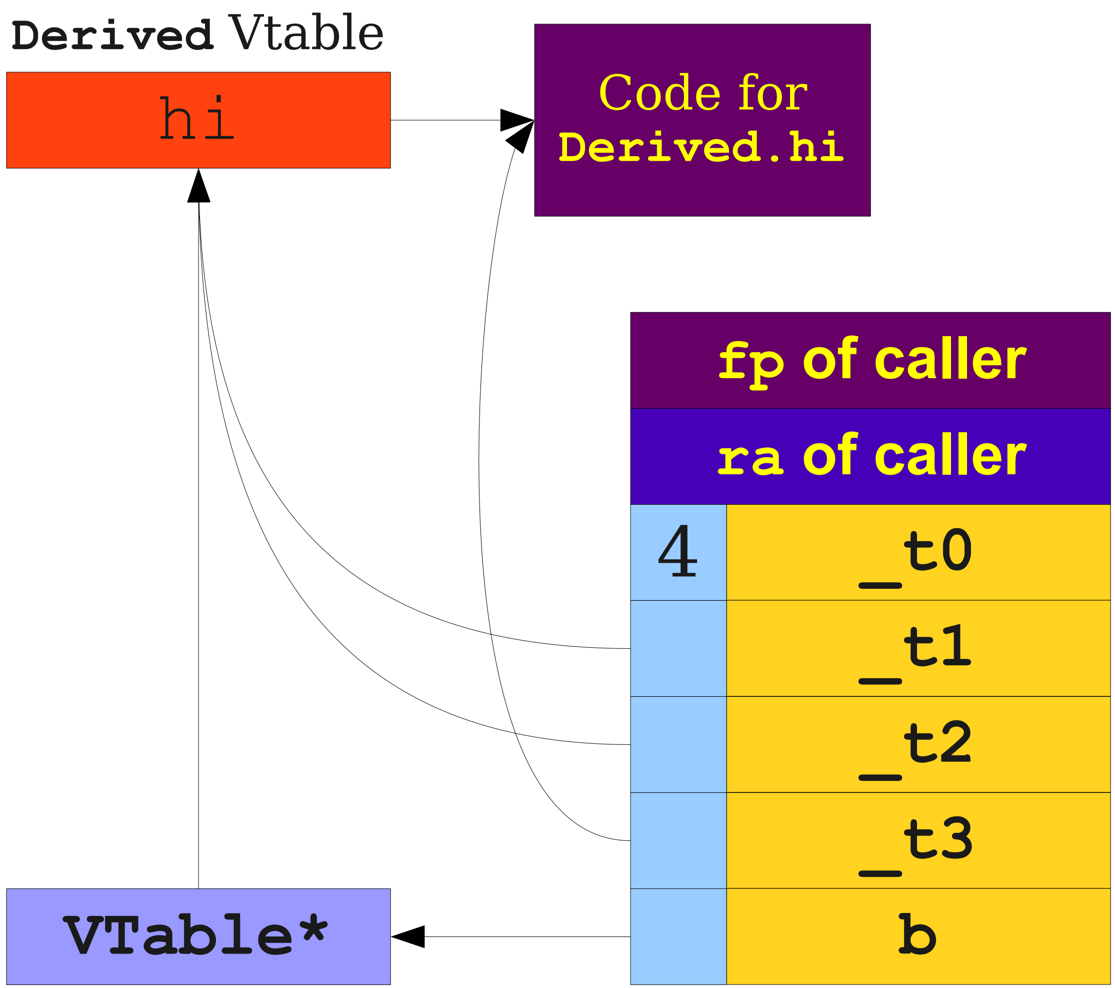

```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Set Vtable

Dissecting TAC



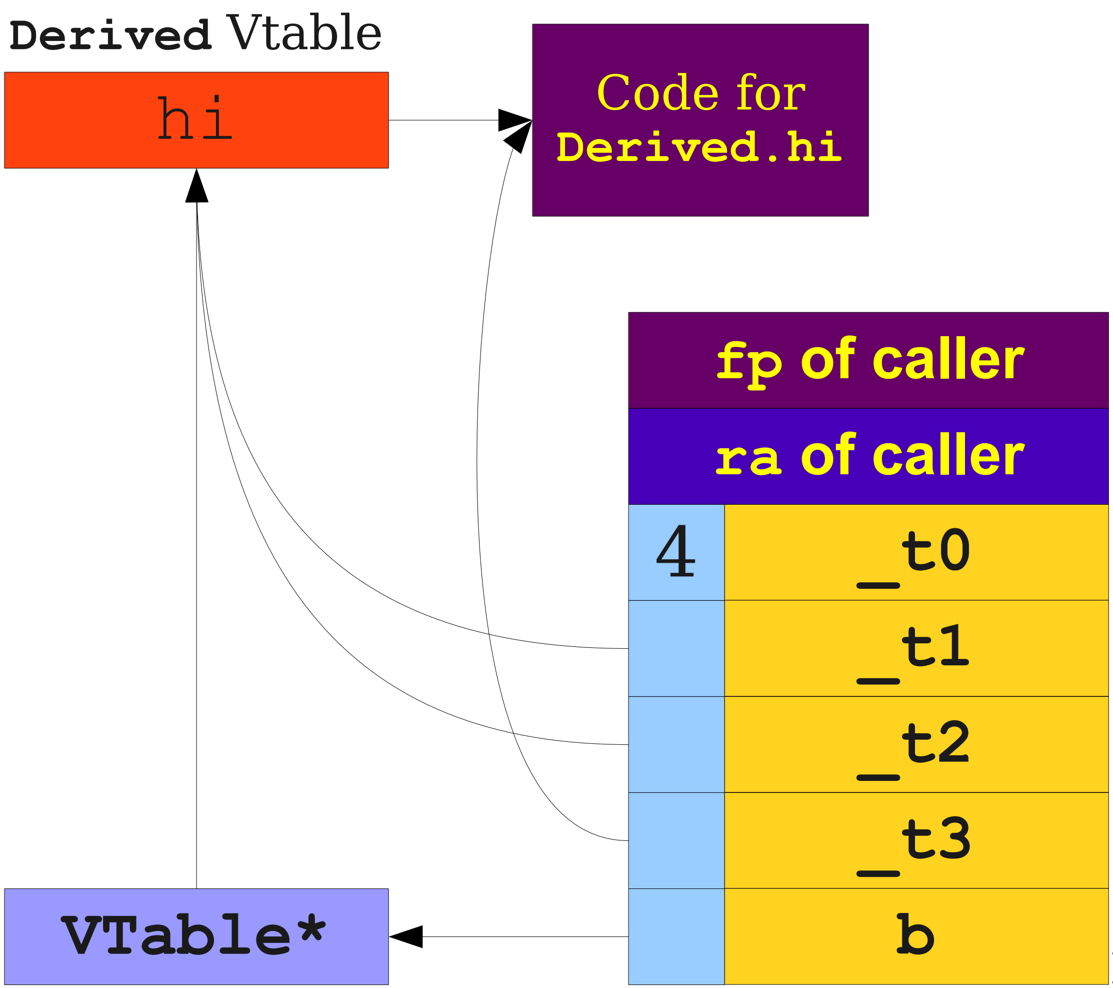

```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Allocate Object

Set Vtable

Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

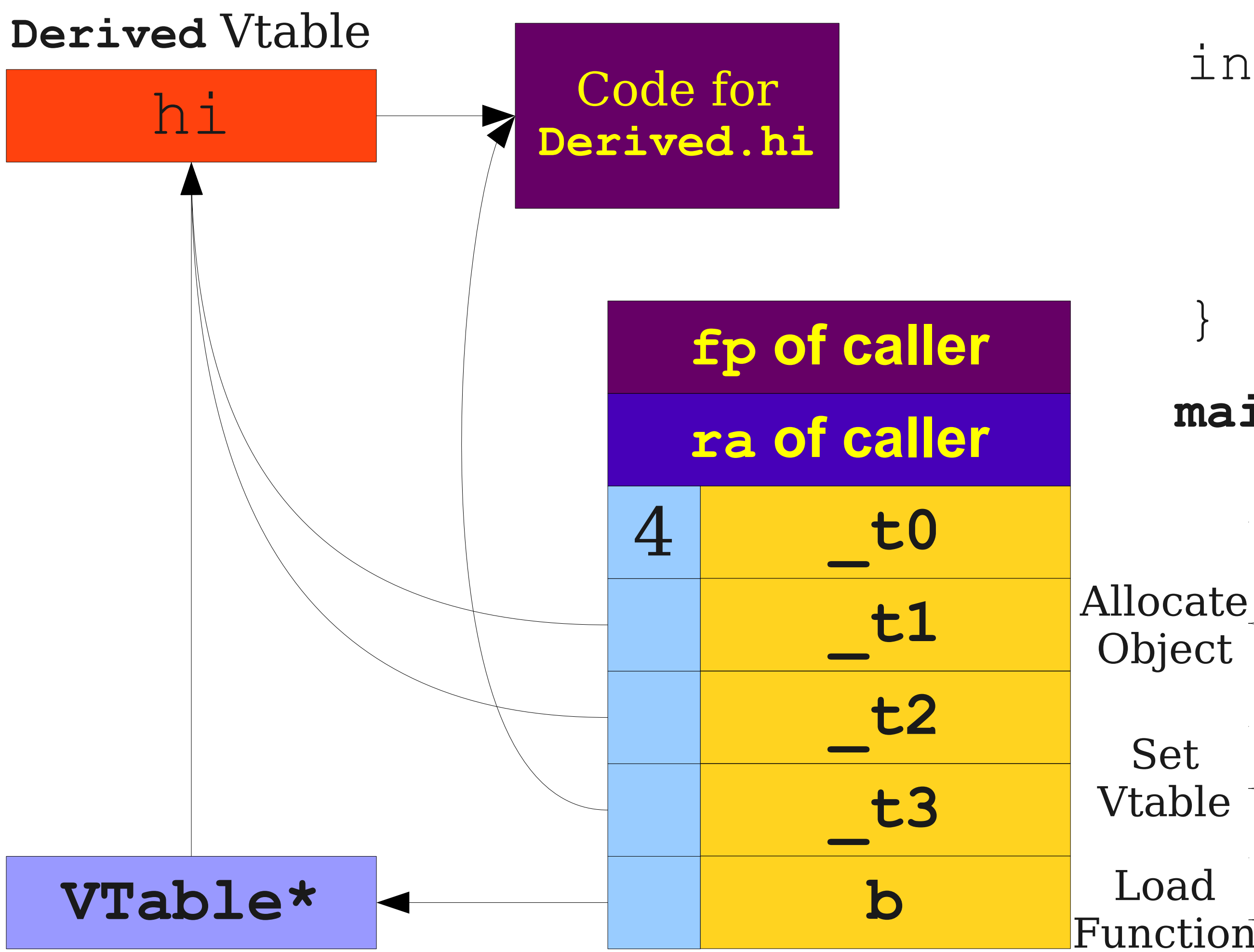
```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Allocate Object

Set Vtable

Load Function

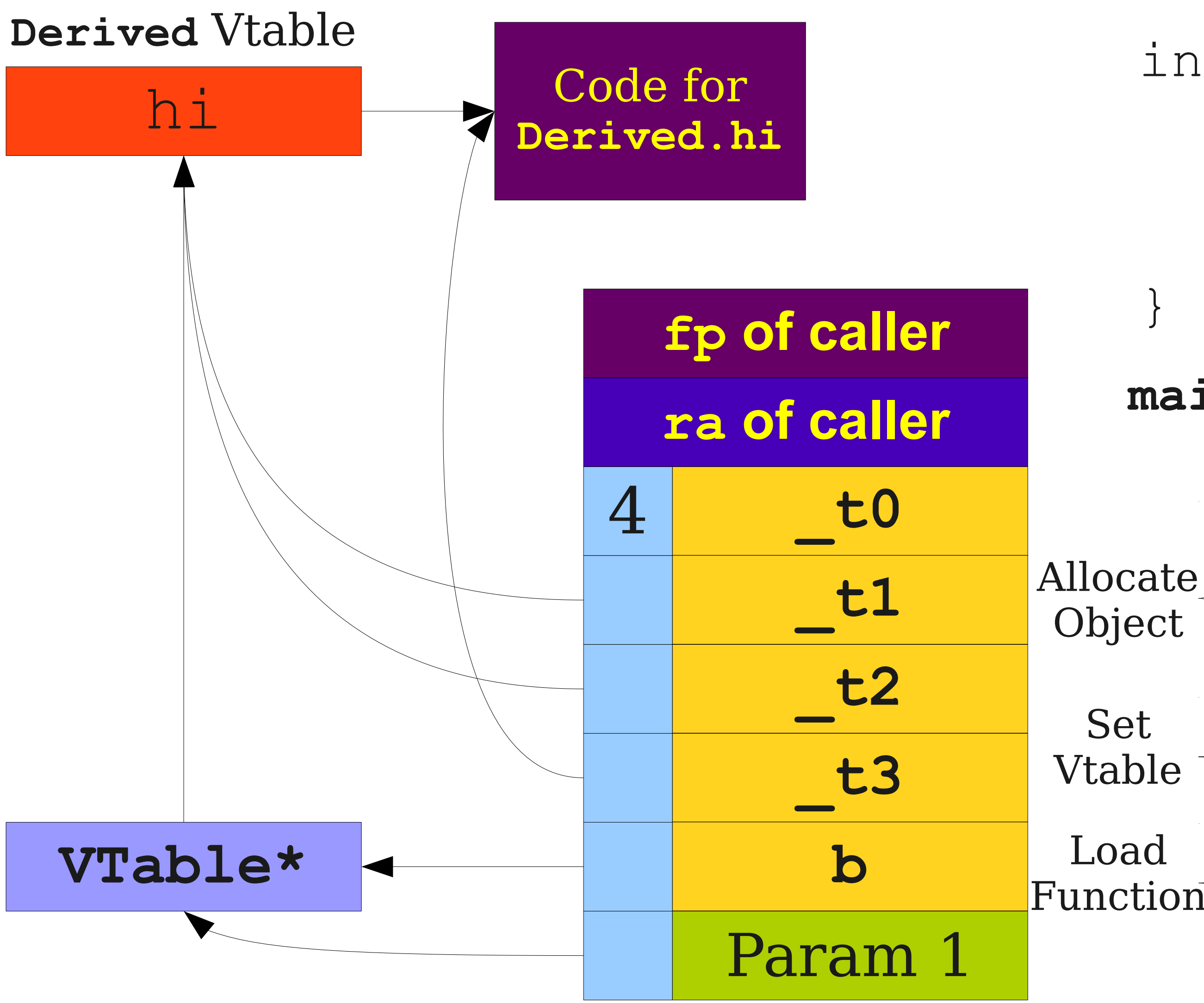
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

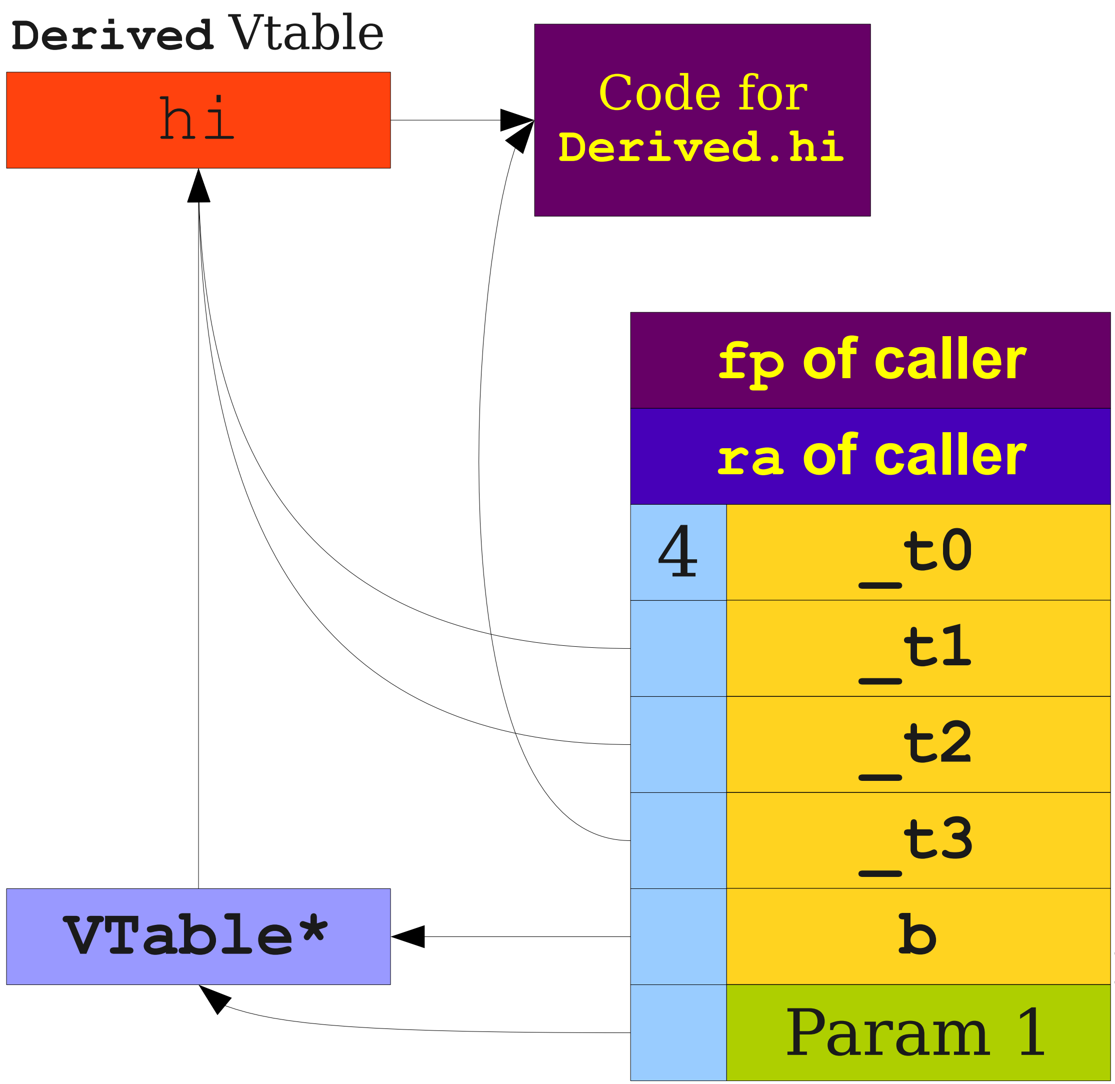
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC




```
int main() {
      Base b;
      b = new Derived;
      b.hi();
}
```

```
main:
      BeginFunc 20;
```

Allocate Object

```
      _t0 = 4;
      PushParam _t0;
      b = LCall _Alloc;
      PopParams 4;
```

Set Vtable

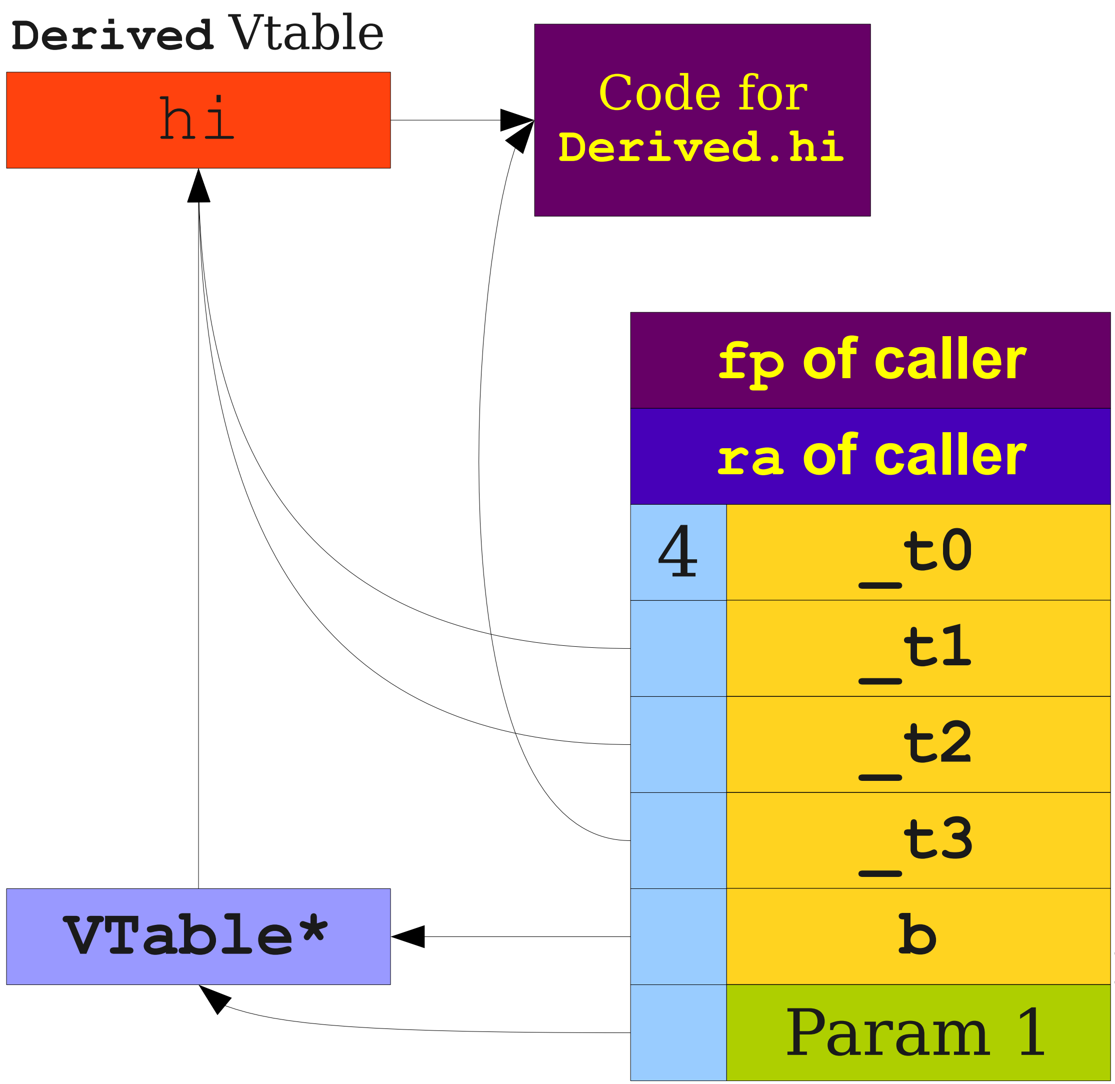
```
      _t1 = Derived;
      *b = _t1;
```

Load Function

```
      _t2 = *b;
      _t3 = *_t2;
```

```
      PushParam b;
      ACall _t3;
      PopParams 4;
      EndFunc;
```

Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

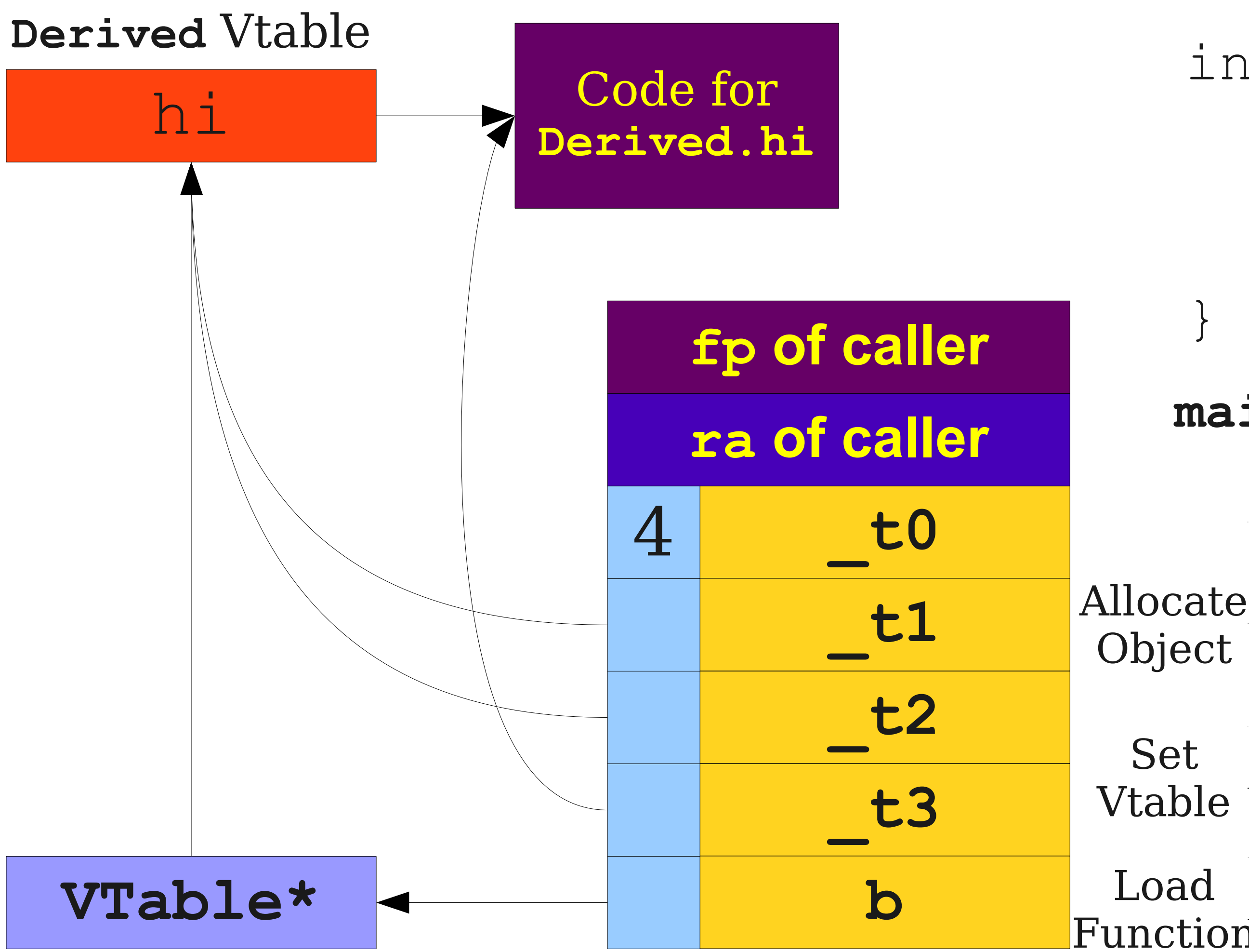
```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Set Vtable

Load Function

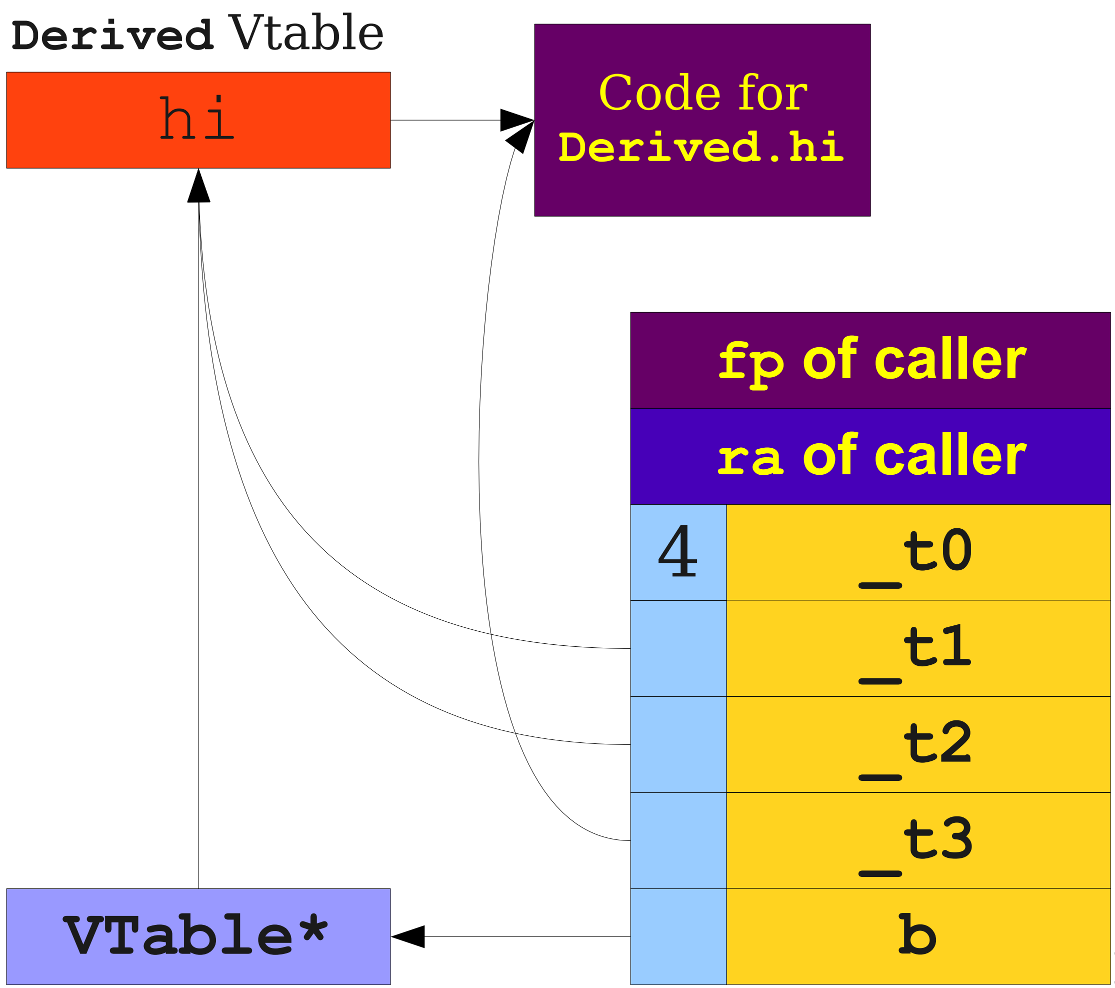
Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

Allocate Object

Set Vtable

Load Function

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

OOP in TAC

- The address of an object's vtable can be referenced via the name assigned to the vtable (usually the object name).
  - e.g. **`_t0 = Base;`**
- When creating objects, you must remember to set the object's vtable pointer or any method call will cause a crash at runtime.
- The **`ACall`** instruction can be used to call a method given a pointer to the first instruction.

Generating TAC

TAC Generation

- At this stage in compilation, we have
  - an AST,
  - annotated with scope information,
  - and annotated with type information.
- To generate TAC for the program, we do (yet another) recursive tree traversal!
  - Generate TAC for any subexpressions or substatements.
  - Using the result, generate TAC for the overall expression.

TAC Generation for Expressions

- Define a function **cgen**(*expr*) that generates TAC that computes an expression, stores it in a temporary variable, then hands back the name of that temporary.
- Define **cgen** directly for atomic expressions (constants, **`this`**, identifiers, etc.).
- Define **cgen** recursively for compound expressions (binary operators, function calls, etc.)

cgen for Basic Expressions

cgen for Basic Expressions

cgen(k) = { // k is a constant

Choose a new temporary t

Emit($t = k$);

Return t

}

cgen for Basic Expressions

```
cgen(k) = { // k is a constant
    Choose a new temporary t
    Emit( t = k );
    Return t
}
cgen(id) = { // id is an identifier
    Choose a new temporary t
    Emit( t = id )
    Return t
}
```

cgen for Binary Operators

cgen for Binary Operators

cgen($e_1 + e_2$) = {

Choose a new temporary t

Let t_1 = **cgen**(e_1)

Let t_2 = **cgen**(e_2)

Emit($t = t_1 + t_2$)

Return t

}

An Example

cgen(5 + x) = {

Choose a new temporary *t*

Let t_1 = **cgen**(5)

Let t_2 = **cgen**(x)

Emit ($t = t_1 + t_2$)

Return t

}

An Example

cgen(5 + x) = {

  Choose a new temporary *t*

  Let t_1 = {

    Choose a new temporary *t*

    Emit(*t* = 5)

    return *t*

  }

  Let t_2 = **cgen**(x)

  Emit (t = t_1 + t_2)

  Return t

}

An Example

cgen(5 + x) = {
Choose a new temporary *t*
Let t_1 = {
Choose a new temporary *t*
Emit(*t* = 5)
return *t*
}
Let t_2 = {
Choose a new temporary *t*
Emit(*t* = x)
return *t*
}
Emit ($t = t_1 + t_2$)
Return t
}

An Example

cgen(5 + x) = {
    Choose a new temporary *t*
    Let t_1 = {
        Choose a new temporary *t*
        Emit(*t* = 5)
        return *t*
    }
    Let t_2 = {
        Choose a new temporary *t*
        Emit(*t* = x)
        return *t*
    }
    Emit (t = t_1 + t_2)
    Return t
}

```
_t0 = 5
_t1 = x
_t2 = _t0 + _t1
```

cgen for Statements

- We can extend the **cgen** function to operate over statements as well.
- Unlike **cgen** for expressions, **cgen** for statements does not return the name of a temporary holding a value.
  - *(Why?)*

cgen for Simple Statements

cgen for Simple Statements

$$\begin{aligned}&\mathbf{cgen}(\mathit{expr};) = \{\\&\qquad \mathbf{cgen}(\mathit{expr})\\&\}\end{aligned}$$

cgen for **`while`** loops

cgen for `while` loops

cgen(`while (`*expr*`)` *stmt*) = {

}

cgen for `while` loops

cgen(`while (`*expr*`)` *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

}

cgen for `while` loops

cgen(`while (`*expr*`)` *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before} :)

Emit(L_{after} :)

}

cgen for `while` loops

cgen(**`while`** **`(`**$expr$**`)`** $stmt$) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before}:)

Let t = **cgen**($expr$)

Emit(**`IfZ`** t **`Goto`** L_{after})

Emit(L_{after}:)

}

cgen for `while` loops

cgen(`while (`*expr*`)` *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before}:)

Let t = **cgen**(*expr*)

Emit(`IfZ` t `Goto` L_{after})

cgen(*stmt*)

Emit(L_{after}:)

}

cgen for `while` loops

cgen(`while (`*expr*`)` *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before}:)

Let t = **cgen**(*expr*)

Emit(`IfZ` t `Goto` L_{after})

cgen(*stmt*)

Emit(`Goto` L_{before})

Emit(L_{after}:)

}

Next Time

- **Intro to IR Optimization**
  - Basic Blocks
  - Control-Flow Graphs
  - Local Optimizations