# Everyday Classes in LLVM

## A whirlwind tour - Nick Desaulniers

# Casting within LLVM

isa<>, cast<>, dyn_cast<>

Google

# Casting

LLVM is full of its own form of RTTI.

```
if (isa<Constant>(foo))
  cast<Constant>(foo)->bar();

if (auto *CI = dyn_cast<CallInst>(I))
  CI->foo();
```

# Casting

Required Reading:

- LLVM Programmer's Manual section on casting
- How to set up LLVM-style RTTI for your class hierarchy
    - TL;DR
        1. Add new enum value to base class.
        2. Implement `static bool classof (const T*)`.

# Casting

Perhaps other interesting reads:

- How Expensive is RTTI? (stackoverflow)
- C++ Tricks: Fast RTTI and Dynamic Cast
- Google C++ Style Guide comments on RTTI
- Casting.h source

# Casting

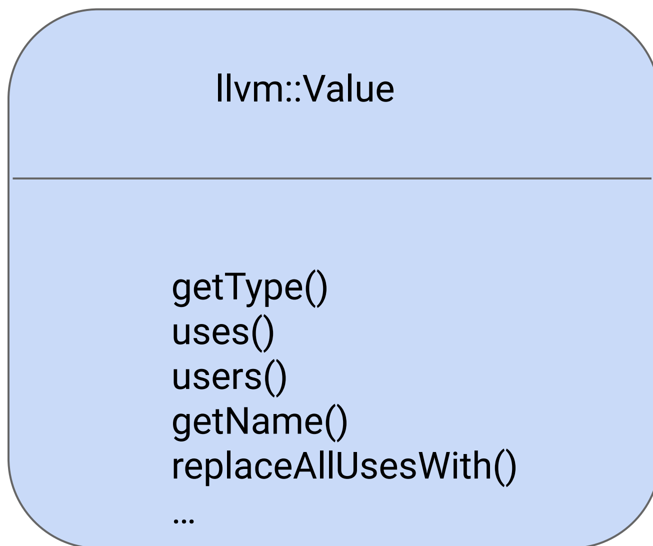
LLVM's RTTI doesn't require vtables, may be more efficient for deep class hierarchies or hierarchies involving multiple inheritance.

isa<>, cast<> dyn_cast<> also have _or_null suffixed equivalents than can handle nullptr arguments.

# Important Classes

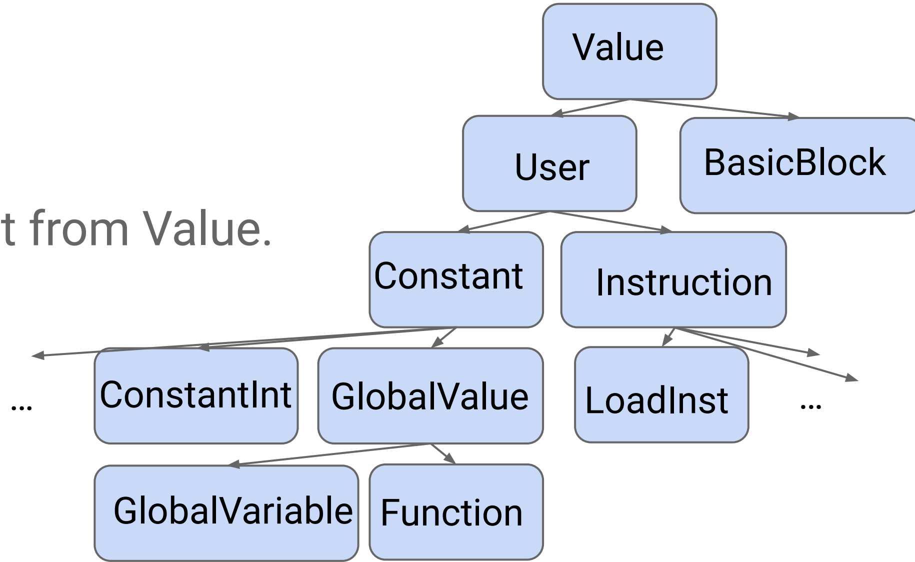
# llvm::Value

llvm::Value is the most important class in the LLVM Source base.



# llvm::Value

Many classes inherit from Value.




See Doxygen for fuller hierarchy.

# User vs Use

- The reason LLVM IR is SSA form.
- https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains
- Given a User, find the Values used aka “use-def” chain.
  - User::operands -> iterator<Value>
- Given a Value, find the Uses aka “def-use” chain.
  - Value::uses() -> iterator<Use>
  - A Use represents the edge between a Value definition and its users.

# User vs Use

```
@.str = private unnamed_addr constant [13 x i8] c"hello world!\00", align 1

define i32 @main(i32 %0, i8** nocapture readnone %1) local_unnamed_addr #0 {
 %3 = tail call i32 @puts(i8* nonnull dereferenceable(1) getelementptr inbounds
([13 x i8], [13 x i8]* @.str, i64 0, i64 0))
 ret i32 %3
}

declare noundef i32 @puts(i8* nocapture noundef readonly) local_unnamed_addr #1
```

# User vs Use

Def

```
@.str = private constant [12 x i8] c"hello world\00"

define i32 @foo() {
 %1 = tail call i32 @puts(ptr @.str)
 ret i32 %1
}

declare i32 @puts(ptr)
```

# User vs Use

```
@.str = private constant [12 x i8] c"hello world\00"

define i32 @foo() {
 %1 = tail call i32 @puts(ptr @.str)
 ret i32 %1
}

declare i32 @puts(ptr)
```

**Def**

**Value V;**
**Use *U = *V.uses();**

**Use**

# User vs Use

```
@.str = private constant [12 x i8] c"hello world\00"

define i32 @foo() {
 %1 = tail call i32 @puts(ptr @.str)
 ret i32 %1
}

declare i32 @puts(ptr)
```

Def

Use U;
Value *Def = U.get();

Use

Google

# User vs Use

```
@.str = private constant [12 x i8] c"hello world\00"

define i32 @foo() {
 %1 = tail call i32 @puts(ptr @.str)
 ret i32 %1
}

declare i32 @puts(ptr)
```

Use U;
User *U2 = U.getUser();

User: @puts

Use: @.str

# User vs Use

```
@.str = private constant [12 x i8] c"hello world\00"

define i32 @foo() {
 %1 = tail call i32 @puts(ptr @.str)
 ret i32 %1
}

declare i32 @puts(ptr)
```

**User U;**

**Use &U2 = *U.operands();**

**User** (@puts)

**Use** (@.str)

Google

# Module, Function, BasicBlock, Instruction

A Module contains 0:N Functions.

```
for (Function &F : Module)
     F.getParent();
```

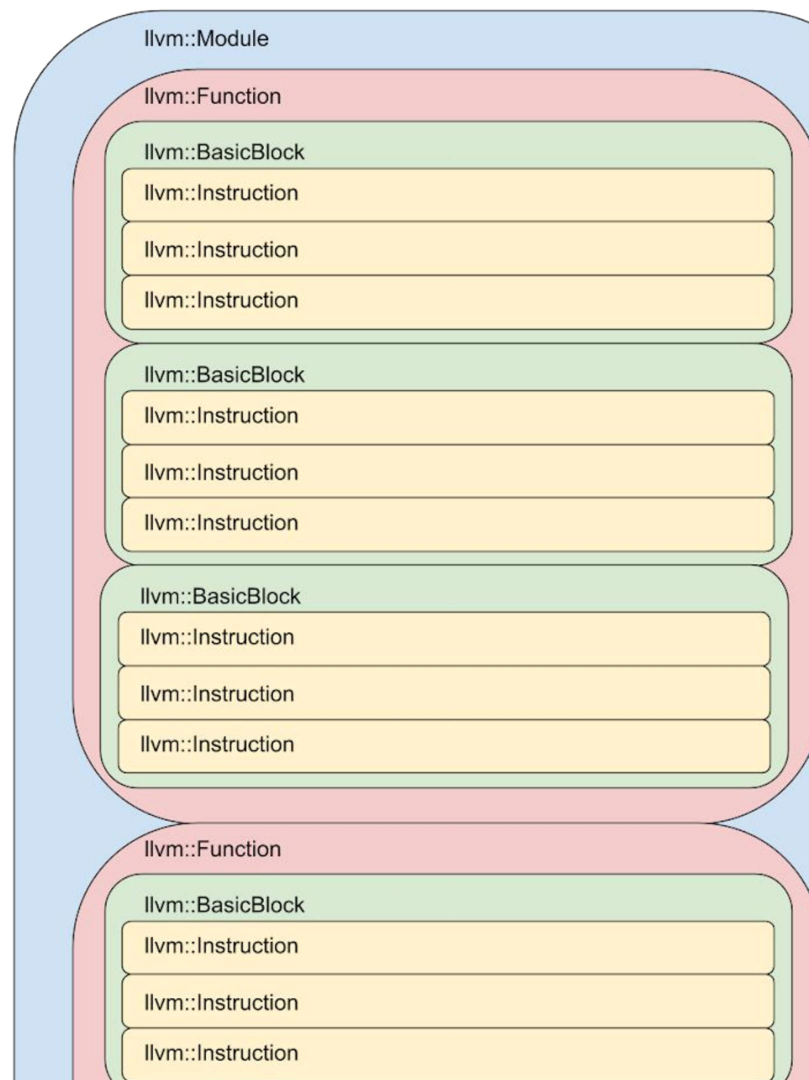
A Function has 1:N BasicBlocks.

```
for (BasicBlock &BB : F)
     BB.getParent();
```

A BasicBlock has 1*:N Instructions.

```
for (Instruction &I : BB)
     I.getParent();
```

Instructions (Users) have 0:N operands.



# BasicBlock

BasicBlocks have 0:N predecessors and 0:N successors (llvm/IR/CFG.h).

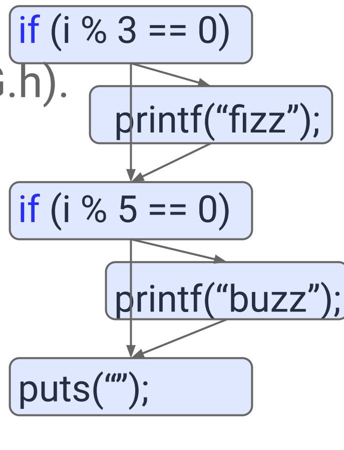
```
for (BasicBlock &Pred : predecessors(BB))
    …
for (BasicBlock &Succ : successors(BB))
    …
```

[https://llvm.org/docs/ProgrammersManual.html#iterating-over-predecessors-successors-of-blocks](https://llvm.org/docs/ProgrammersManual.html#iterating-over-predecessors-successors-of-blocks)

BasicBlocks have 1* terminating Instruction.

```
Instruction *Term = BB.getTerminator();
```

void fizzbuzz (int i) {




}

Google

# SelectionDAG

SDValue

SDNode

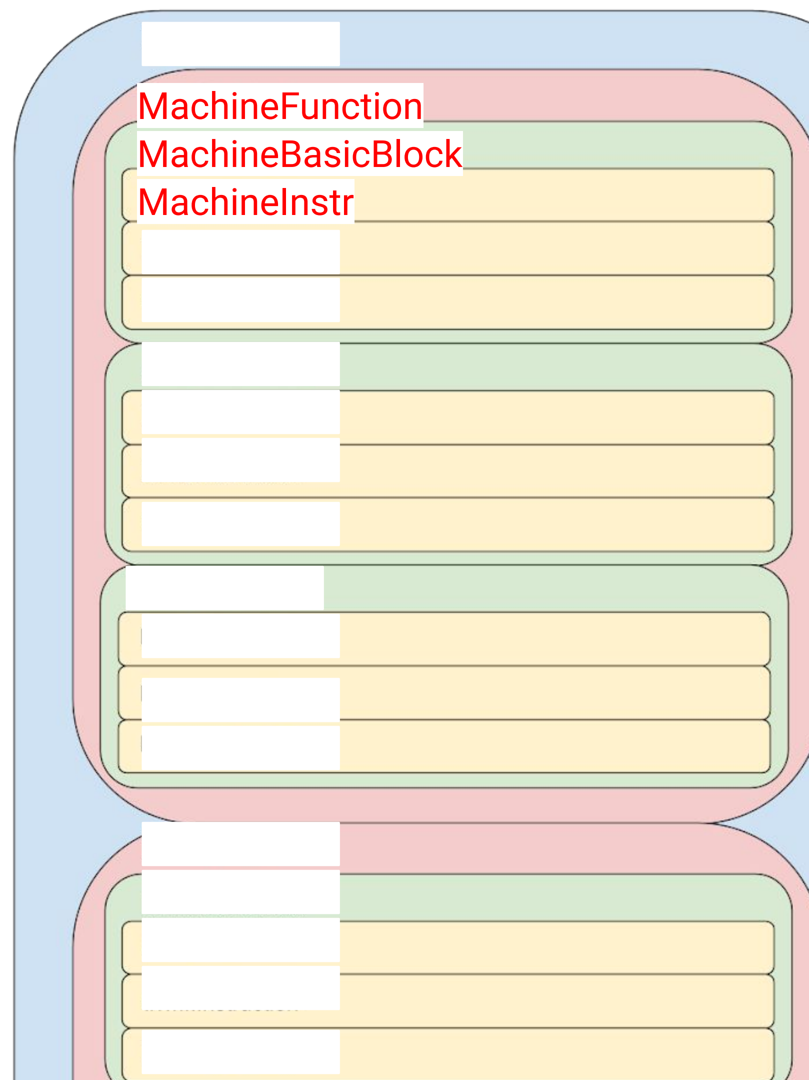
An SDNode has 1:N SDValues.

SelectionDAGBuilder -> DAGCombiner -> DAGTypeLegalizer -> DAGCombiner -> SelectionDAGLegalize -> SelectionDAGISel -> ScheduleDAGSDNodes

https://llvm.org/docs/CodeGenerator.html#selectiondag-instruction-selection-process

# MIR: MachineFunction, MachineBasicBlock, MachineInstr

- Similar to Function, BasicBlock, and Instruction.
- No Module equivalent.
- MachineFunction has a pointer back to the corresponding Function.
- Similar iterators.
- MachineBasicBlocks may have multiple terminators!
- MachineInstr may have virtual register operands.



# MIR: MachineFunction, MachineBasicBlock, MachineInstr

- Similar to Function, BasicBlock, and Instruction.
- Looks a little different though!

C

```c
#include <stdio.h>

int main (int argc, char** argv) {
 puts("hello world!");
}
```

# LLVM IR

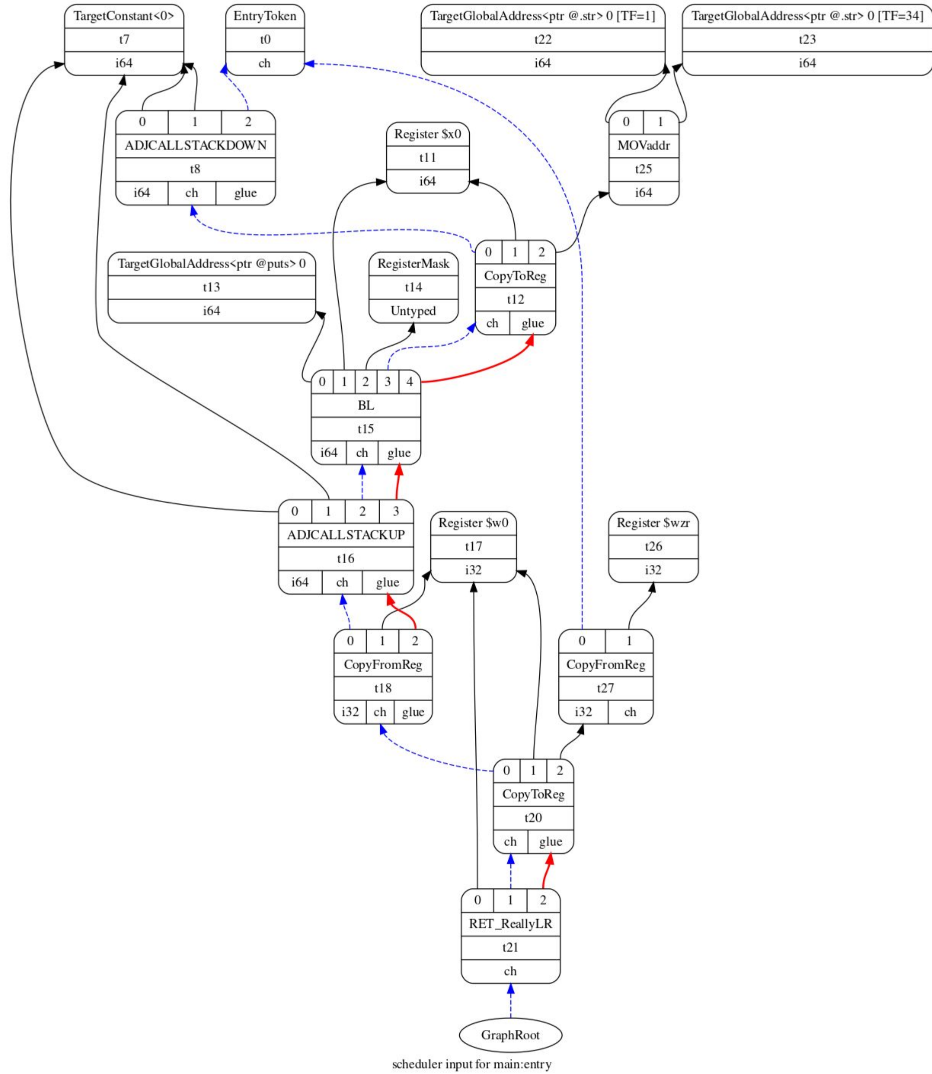
```
@.str = private unnamed_addr constant [13 x i8] c"hello world!\00", align 1

define i32 @main(i32 %0, i8** nocapture readnone %1) local_unnamed_addr #0 {
 %3 = tail call i32 @puts(i8* nonnull dereferenceable(1) getelementptr inbounds
([13 x i8], [13 x i8]* @.str, i64 0, i64 0))
 ret i32 0
}

declare noundef i32 @puts(i8* nocapture noundef readonly) local_unnamed_addr #1
```

Google

# SelectionDAG




Google

# MIR (dont try to write this by hand; llc -dump-before=<pass>)

```
Frame Objects:
  fi#0: size=8, align=8, at location [SP-8]
  fi#1: size=8, align=8, at location [SP-16]

bb.0 (%ir-block.2):
  liveins: $lr
  early-clobber $sp = frame-setup STPXpre killed $fp, killed $lr, $sp(tied-def 0), -2 :: (store (s64) into %stack.1), (store (s64) into %stack.0)
  $fp = frame-setup ADDXri $sp, 0, 0
  $x0 = ADRP target-flags(aarch64-page) @.str
  renamable $x0 = ADDXri $x0, target-flags(aarch64-pageoff, aarch64-nc) @.str, 0
  BL @puts, <regmask $fp $lr $b8 $b9 $b10 $b11 $b12 $b13 $b14 $b15 $d8 $d9 $d10 $d11 $d12 $d13 $d14 $d15 $h8 $h9 $h10 $h11 $h12 $h13 $h14 $h15 $s8 $s9 $s10 $s11 $s12 $s13 $s14 and 53 more...>, implicit-def dead $lr, implicit $sp, implicit $x0, implicit-def $sp, implicit-def dead $w0
  $w0 = MOVZWi 0, 0
  early-clobber $sp, $fp, $lr = frame-destroy LDPXpost $sp(tied-def 0), 2 :: (load (s64) from %stack.1), (load (s64) from %stack.0)
  RET undef $lr, implicit $w0
```

Google

# MIR (dont try to write this by hand; llc -dump-before=<pass>)

```
Frame Objects:
  fi#0: size=8, align=8, at location [SP-8]
  fi#1: size=8, align=8, at location [SP-16]

bb.0 (%ir-block.2):
  liveins: $lr
  early-clobber $sp = frame-setup STPXpre killed $fp,
(store (s64) into %stack.0)
  $fp = frame-setup ADDXri $sp, 0, 0
  $x0 = ADRP target-flags(aarch64-page) @.str
  renamable $x0 = ADDXri $x0, target-flags(aarch64-p
  BL @puts, <regmask $fp $lr $b8 $b9 $b10 $b11 $b1
$h8 $h9 $h10 $h11 $h12 $h13 $h14 $h15 $s8 $s9 $s
$lr, implicit $sp, implicit $x0, implicit-def $sp, implicit
  $w0 = MOVZWi 0, 0
  early-clobber $sp, $fp, $lr = frame-destroy LDPXpos
from %stack.0)
  RET undef $lr, implicit $w0
```

```
_main:
    stp x29, x30, [sp, #-16]!
    mov x29, sp
    adrp    x0, l_.str@PAGE
    add x0, x0, l_.str@PAGEOFF
    bl  _puts
    mov w0, #0
    ldp x29, x30, [sp], #16
    ret
```

# DominatorTree

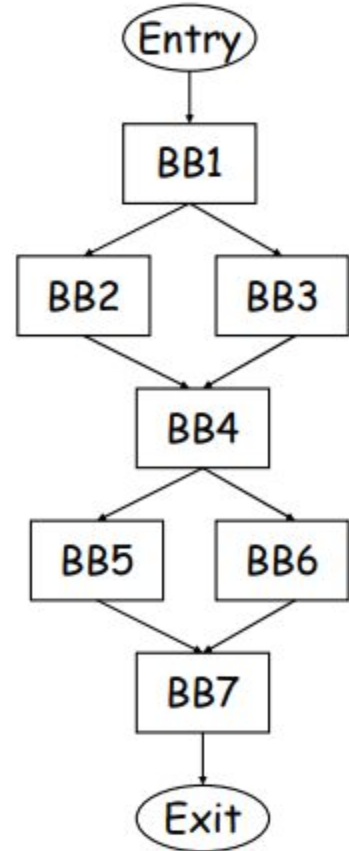
DominatorTree::dominates(const Value *Def, const Use &User)

A node *d* of a control-flow graph dominates a node *n* if every path from the entry node to *n* must go through *d*.

Does BB 1 dominate BB 7?

Does BB 7 dominate BB 4?

Does BB 5 dominate BB 7?



# DominatorTree

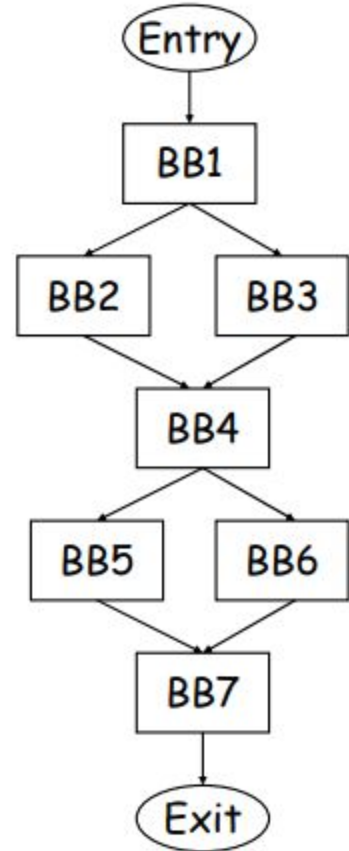
DominatorTree::dominates(const Value *Def, const Use &User)

A node *d* of a control-flow graph dominates a node *n* if every path from the entry node to *n* must go through *d*.

Does BB 1 dominate BB 7? d:1 n:7 Yes

Does BB 7 dominate BB 4? d:7 n:4 No (wrong direction)

Does BB 5 dominate BB 7? d:5 n:7 No



# Containers

# ADT

- Refer to "Picking the Right Data Structure for a Task" section of LLVM Programmer's Manual.
- Vectors appear frequently for worklists.
    - `llvm::SmallVector` implements "small string-like" optimizations.
- Pretty easy to "dunk on" `std::unordered_map`. Prefer `llvm::DenseMap`.
- Many containers have variants when the key or value is either a pointer or a string.
    - SmallPtrSet
    - StringSet
    - StringMap
- Sometime std lib is the way to go; read the programmers manual.

# StringRef, ArrayRef, & Twine

- StringRef and ArrayRef used frequently for function interfaces.
    - C string literals, std::string, char arrays, llvm::SmallVector are convertible to StringRef.
    - Arrays, std::vector, llvm::SmallVector are convertible to ArrayRef.
    - Remind me of std::span; they don't own the memory they refer to, so watch for -Wdangling-gsl!
- C++ std lib missing some useful string functions; wrap in StringRef for helpers.
- Building up a string via repeated concatenations is not efficient from a memory allocation perspective, or requires std::stringstream. Enter llvm::Twine.

# APInt (Arbitrary precision integers)

```
APInt(unsigned numBits, uint64_t val, bool isSigned = false)
```

```
uint64_t val = MyAPInt.getZExtValue();
```

```
int64_t sval = MyAPInt.getSExtValue();
```

- Other constructors available for val > ULONG_MAX.
- Class implements operators you'd expect, helper functions, static factory methods.

# Passes

# PassBuilder

```
buildPerModuleDefaultPipeline();

buildPerModuleDefaultPipeline(OptimizationLevel::O2);
```

# Passes

```
Pass::doInitialization(Module &);

Pass::doFinalization(Module &);

ModulePass::runOnModule(Module &);

FunctionPass::runOnFunction(Function &);
```

# Clang

# Decl & Expr

**$ clang -Xclang -ast-dump foo.c**

```
`-FunctionDecl 0x1310c8290 </tmp/x.c:2:1, line:4:1> line:2:5 main 'int (int, char **)'
  |-ParmVarDecl 0x1310c8138 <col:11, col:15> col:15 argc 'int'
  |-ParmVarDecl 0x1310c81b8 <col:21, col:28> col:28 argv 'char **'
  `-CompoundStmt 0x1310c8460 <col:34, line:4:1>
    `-CallExpr 0x1310c8408 <line:3:3, col:22> 'int'
      |-ImplicitCastExpr 0x1310c83f0 <col:3> 'int (*)(const char *)' <FunctionToPointerDecay>
      | `-DeclRefExpr 0x1310c8340 <col:3> 'int (const char *)' Function 0x131060a30 'puts' 'int (const char *)'
      `-ImplicitCastExpr 0x1310c8448 <col:8> 'const char *' <NoOp>
        `-ImplicitCastExpr 0x1310c8430 <col:8> 'char *' <ArrayToPointerDecay>
          `-StringLiteral 0x1310c8398 <col:8> 'char [13]' lvalue "hello world!"
```

# Decl & Expr

Subclasses of Decl

$ clang -Xclang -ast-dump foo.c

```
`-FunctionDecl 0x1310c8290 </tmp/x.c:2:1, line:4:1> line:2:5 main 'int (int, char **)'
  |-ParmVarDecl 0x1310c8138 <col:11, col:15> col:15 argc 'int'
  |-ParmVarDecl 0x1310c81b8 <col:21, col:28> col:28 argv 'char **'
  `-CompoundStmt 0x1310c8460 <col:34, line:4:1>
    `-CallExpr 0x1310c8408 <line:3:3, col:22> 'int'
      |-ImplicitCastExpr 0x1310c83f0 <col:3> 'int (*)(const char *)' <FunctionToPointerDecay>
      | `-DeclRefExpr 0x1310c8340 <col:3> 'int (const char *)' Function 0x131060a30 'puts' 'int (const char *)'
      `-ImplicitCastExpr 0x1310c8448 <col:8> 'const char *' <NoOp>
        `-ImplicitCastExpr 0x1310c8430 <col:8> 'char *' <ArrayToPointerDecay>
          `-StringLiteral 0x1310c8398 <col:8> 'char [13]' lvalue "hello world!"
```

Subclasses of Expr

# Sema & Diag & DiagnosticsEngine & SourceLocation

```
Diag(Loc, diag::warn_foo) << Expr << 10;

DiagnosticsEngine::IsIgnored(diag::warn_foo, Loc)
```

# LangOptions

LangOpts.CPlusPlus || LangOpts.GNUMode

clang/include/clang/Basic/LangOptions.def

Google

# Recommended Reading

# For more info

- Learn LLVM 12: A beginner's guide to learning LLVM compiler tools and core libraries with C++ - Kai Nacke
- LLVM Techniques, Tips, and Best Practices: Clang and Middle-End Libraries: Design powerful and reliable compilers using the latest libraries and tools from LLVM - Min-Yih Hsu
- Kaleidoscope: Implementing a Language with LLVM
- Full class list