TypeLayout

Complete Tutorial

TypeLayout Team

Version 1.0, 2026-02-03: Initial Release

Table of Contents

A comprehensive guide to type layout verification in C++26 using static reflection.

Chapter 1. Preface

TypeLayout is a modern C++26 header-only library that generates compile-time type layout signatures using P2996 static reflection. It addresses the critical challenge of binary compatibility verification across different platforms, compilers, and code versions.

This tutorial provides a progressive learning path—from understanding the fundamental problem to mastering advanced usage patterns.

Target Audience:

- C++ developers working with binary data formats
- Systems programmers dealing with cross-platform compatibility
- Game developers managing save files and network protocols
- Library maintainers concerned about ABI stability

Prerequisites:

- Solid understanding of C++ structs and memory layout
- Familiarity with concepts like `sizeof`, `alignof`, and padding
- Access to a C++26 compiler with P2996 support (Bloomberg's experimental Clang)

What You'll Learn:

- Why struct layout causes subtle, hard-to-debug bugs
- How to generate and read layout signatures
- Practical applications in IPC, networking, and file formats
- Advanced features: inheritance, polymorphism, bitfields
- How P2996 reflection works under the hood

Chapter 2. Chapter 1: The Hidden Bug

Why struct layout matters and the subtle bugs it can cause.

2.1. A Simple Configuration File

Imagine you're building a game that saves player configuration to a file:

```
// game_config.hpp
#include <cstdint>
#include <fstream>

struct GameConfig {
    int32_t  version;      // Config file version
    bool     fullscreen;   // Fullscreen mode?
    int64_t  last_played;  // Unix timestamp
    uint32_t volume;       // 0-100
};

void save_config(const GameConfig& cfg, const char* path) {
    std::ofstream file(path, std::ios::binary);
    file.write(reinterpret_cast<const char*>(&cfg), sizeof(cfg));
}

GameConfig load_config(const char* path) {
    GameConfig cfg{};  // Zero-initialize
    std::ifstream file(path, std::ios::binary);
    file.read(reinterpret_cast<char*>(&cfg), sizeof(cfg));
    return cfg;
}
```

This code looks correct. It compiles without warnings. It works on your machine.

But there's a hidden bug.

2.2. The Bug Revealed

Let's examine the actual memory layout:

```
struct GameConfig {
    int32_t  version;      // offset 0, size 4
    bool     fullscreen;   // offset 4, size 1
    // 3 bytes padding here! (for int64_t alignment)
    int64_t  last_played;  // offset 8, size 8
    uint32_t volume;       // offset 16, size 4
    // 4 bytes padding here! (for struct alignment)
};
```

```
// Total size: 24 bytes, alignment: 8
```

The struct has 7 **bytes of hidden padding**. Now consider these scenarios:

2.2.1. Scenario 1: Cross-Platform Sharing

Platform	`long` size	`bool` alignment
Linux x86_64 (LP64)	8 bytes	1 byte
Windows x64 (LLP64)	4 bytes	1 byte
Some ARM platforms	4 bytes	4 bytes

If a Linux server saves the config and a Windows client loads it… **data corruption**.

2.2.2. Scenario 2: Compiler Differences

Even on the same OS, different compilers may use different padding strategies:

- GCC vs Clang vs MSVC
- Debug vs Release builds
- Different optimization flags

2.2.3. Scenario 3: Code Evolution

A future developer adds a new field:

```
struct GameConfig {
    int32_t  version;
    bool     fullscreen;
    bool     vsync;        // NEW: VSync enabled?
    int64_t  last_played;
    uint32_t volume;
};
```

Old config files are now **silently incompatible**. No error, just wrong values.

2.3. The Real-World Impact

These bugs are:

- **Silent** - No compiler warning, no runtime error
- **Intermittent** - Works on some platforms, fails on others
- **Hard to debug** - Data looks almost right but is slightly shifted

Common victims:

- Game save files
- Configuration persistence
- Shared memory between processes
- Network protocols
- Memory-mapped files

2.4. What We Need

We need a way to:

1. **Capture** the exact memory layout of a struct at compile time
2. **Compare** layouts to detect incompatibilities before they cause bugs
3. **Verify** that layouts match expected specifications
4. **Do it all at compile time** - catch bugs before deployment

This is exactly what TypeLayout provides.

Chapter 3. Chapter 2: The Old Ways

Traditional solutions to struct layout problems and their limitations.

3.1. Solution 1: Manual static_assert

The most common approach is manual offset verification:

```
struct GameConfig {
    int32_t  version;
    bool     fullscreen;
    int64_t  last_played;
    uint32_t volume;
};

// Manual verification
static_assert(sizeof(GameConfig) == 24);
static_assert(offsetof(GameConfig, version) == 0);
static_assert(offsetof(GameConfig, fullscreen) == 4);
static_assert(offsetof(GameConfig, last_played) == 8);
static_assert(offsetof(GameConfig, volume) == 16);
```

3.1.1. Limitations

Problem	Impact
Tedious	Must write one assertion per field
Error-prone	Easy to forget a field or make typos
Maintenance burden	Must update when struct changes
No private members	`offsetof` fails on non-standard-layout types
No inheritance	Cannot verify base class offsets

3.2. Solution 2: External Tools

Tools like `pahole` and `dwarfdump` can analyze compiled binaries:

```
$ pahole -C GameConfig my_program
struct GameConfig {
    int32_t    version;          /*     0     4 */
    _Bool      fullscreen;       /*     4     1 */
    /* 3 bytes hole */
    int64_t    last_played;      /*     8     8 */
    uint32_t   volume;           /*    16     4 */
    /* 4 bytes padding */
    /* size: 24, alignment: 8 */
};
```

3.2.1. Limitations

Problem	Impact
Post-compilation	Only works after building
External dependency	Not part of C++ build process
Manual inspection	Must compare outputs yourself
No CI/CD integration	Hard to automate verification

3.3. Solution 3: Pragma Pack

Force specific alignment with compiler directives:

```
#pragma pack(push, 1)
struct GameConfig {
    int32_t  version;
    bool     fullscreen;
    int64_t  last_played;
    uint32_t volume;
};
#pragma pack(pop)
// Now size is exactly 17 bytes, no padding
```

3.3.1. Limitations

Problem	Impact
Performance penalty	Unaligned access is slower
Portability issues	Non-standard, varies by compiler
Invasive	Changes the actual layout
Alignment requirements	May violate CPU alignment needs

3.4. Solution 4: Serialization Libraries

Use protobuf, flatbuffers, or similar:

```
// config.proto
message GameConfig {
    int32 version = 1;
    bool fullscreen = 2;
    int64 last_played = 3;
    uint32 volume = 4;
}
```

3.4.1. Limitations

Problem	Impact
IDL overhead	Separate definition language
Generated code	Not native C++ structs
Runtime cost	Serialization/deserialization overhead
Learning curve	New tools and concepts to learn

3.5. What We Really Need

An ideal solution would:

Feature	Manual	External	Pack	IDL
Compile-time verification	Yes	No	No	No
Zero runtime overhead	Yes	Yes	No	No
Native C++ structs	Yes	Yes	Yes	No
Automatic & complete	No	No	Yes	Yes
Private member support	No	Yes	Yes	N/A
Inheritance support	No	Yes	Yes	No

TypeLayout achieves **all of these** using C++26 static reflection.

Chapter 4. Chapter 3: Quick Start

Your first layout signature with TypeLayout.

4.1. Installation

TypeLayout is a header-only library. Simply include the header:

```cpp
#include <boost/typelayout.hpp>
using namespace boost::typelayout;
```

Requires a C++26 compiler with P2996 static reflection support.

4.2. Your First Signature

Let’s generate a layout signature for a simple struct:

```cpp
#include <boost/typelayout.hpp>
#include <cstdint>
#include <iostream>

using namespace boost::typelayout;

struct Point {
    int32_t x;
    int32_t y;
};

int main() {
    // Generate signature at compile time
    constexpr auto sig = get_layout_signature<Point>();

    // Print it (runtime output for inspection)
    std::cout << "Point signature: " << sig.c_str() << '\n';

    return 0;
}
```

Output:

```
Point signature: [64-le]struct[s:8,a:4]{@0[x]:i32[s:4,a:4],@4[y]:i32[s:4,a:4]}
```

4.3. Understanding the Signature

Let’s break down what this signature tells us:

```
[64-le]struct[s:8,a:4]{@0[x]:i32[s:4,a:4],@4[y]:i32[s:4,a:4]}
 │        │      │    │    │  │  │    │          │
 │        │      │    │    │  │  │    │          └── Second field info
 │        │      │    │    │  │  │    └── Type signature
 │        │      │    │    │  │  └── Field name
 │        │      │    │    │  └── Offset in bytes
 │        │      │    │    └── Field list start
 │        │      │    └── Alignment
 │        │      └── Size
 │        └── Type category
 └── Platform prefix
```

4.4. Compile-Time Verification

The real power is compile-time verification:

```
struct Point {
    int32_t x;
    int32_t y;
};

// Verify layout at compile time - fails if layout changes
static_assert(get_layout_signature<Point>() ==
    "[64-le]struct[s:8,a:4]{@0[x]:i32[s:4,a:4],@4[y]:i32[s:4,a:4]}");

// Or use the convenient macro
BOOST_TYPELAYOUT_ASSERT(Point,
    "[64-le]struct[s:8,a:4]{@0[x]:i32[s:4,a:4],@4[y]:i32[s:4,a:4]}");
```

If anyone changes `Point` (even accidentally), compilation fails immediately.

4.5. Comparing Types

Check if two types have the same memory layout:

```
struct PointA { int32_t x, y; };
struct PointB { int32_t a, b; };  // Different names, same layout
struct PointC { int64_t x, y; };  // Different types

// These pass
static_assert(LayoutCompatible<PointA, PointA>);  // Same type
// Note: PointA and PointB have different field names, so not compatible
```

```cpp
static_assert(!LayoutCompatible<PointA, PointB>); // Different field names

// This fails (different sizes)
static_assert(!LayoutCompatible<PointA, PointC>);
```

4.6. Using Hash for Quick Checks

For runtime comparison, use the 64-bit hash:

```cpp
constexpr auto hash = get_layout_hash<Point>();
// Returns a 64-bit FNV-1a hash of the signature

// Useful in headers, IPC, shared memory:
struct SharedMemoryHeader {
    uint64_t layout_hash;  // Store hash for verification
    // ... data ...
};
```

4.7. Exercise: Try It Yourself

1. Create a struct with padding:

   ```cpp
   struct Padded {
       int8_t  a;
       int32_t b;
   };
   ```

2. Generate its signature
3. Predict the size and offsets before checking
4. Verify your predictions with `static_assert`

Chapter 5. Chapter 4: Reading Signatures

Understanding the layout signature format in detail.

5.1. Signature Grammar

A layout signature follows this structure:

```
<signature> ::= <platform> <type-sig>

<platform>  ::= "[" <bits> "-" <endian> "]"
<bits>      ::= "32" | "64"
<endian>    ::= "le" | "be"

<type-sig>  ::= <primitive> | <struct-sig> | <class-sig> |
                <union-sig> | <enum-sig> | <array-sig> | ...
```

5.2. Platform Prefix

The platform prefix identifies the build environment:

Prefix	Meaning
`[64-le]`	64-bit, little-endian (x86_64, ARM64)
`[64-be]`	64-bit, big-endian (POWER, SPARC)
`[32-le]`	32-bit, little-endian (x86)
`[32-be]`	32-bit, big-endian (older ARM)

5.3. Primitive Types

Type	Signature	Size	Notes
`int8_t`	`i8[s:1,a:1]`	1	Signed 8-bit
`uint16_t`	`u16[s:2,a:2]`	2	Unsigned 16-bit
`int32_t`	`i32[s:4,a:4]`	4	Signed 32-bit
`int64_t`	`i64[s:8,a:8]`	8	Signed 64-bit
`float`	`f32[s:4,a:4]`	4	IEEE 754 single
`double`	`f64[s:8,a:8]`	8	IEEE 754 double
`bool`	`bool[s:1,a:1]`	1	Boolean
`char`	`char[s:1,a:1]`	1	Character
`void*`	`ptr[s:8,a:8]`	4/8	Platform-dependent

5.4. Struct Signatures

5.4.1. Basic Format

```
struct[s:SIZE,a:ALIGN]{FIELDS}
```

Example:

```
struct Point { int32_t x, y; };
// [64-le]struct[s:8,a:4]{@0[x]:i32[s:4,a:4],@4[y]:i32[s:4,a:4]}
```

5.4.2. Field Format

Each field has:

```
@OFFSET[NAME]:TYPE
```

- `@OFFSET` - Byte offset from struct start
- `[NAME]` - Field name
- `TYPE` - Field type signature

5.4.3. With Padding

```
struct Padded {
    int8_t  a;   // @0
    // 3 bytes padding (for int32_t alignment)
    int32_t b;   // @4
};
// struct[s:8,a:4]{@0[a]:i8[s:1,a:1],@4[b]:i32[s:4,a:4]}
```

The padding is **implicit** - inferred from the gap between offsets.

5.5. Class Signatures

Classes with inheritance or polymorphism use the `class` keyword:

5.5.1. With Inheritance

```
struct Base { uint64_t id; };
struct Derived : Base { uint32_t value; };
// class[s:16,a:8,inherited]{@0[base]:struct[s:8,a:8]{...},@8[value]:u32[s:4,a:4]}
```

The `inherited` marker indicates the class has base classes.

5.5.2. Polymorphic Classes

```
class Entity {
    virtual ~Entity() = default;
    uint32_t id_;
};
// class[s:16,a:8,polymorphic]{@8[id_]:u32[s:4,a:4]}
```

The `polymorphic` marker indicates a vtable pointer exists (offset 0-7).

5.6. Union Signatures

All union members share offset 0:

```
union Value {
    int32_t i;
    float   f;
};
// union[s:4,a:4]{@0[i]:i32[s:4,a:4],@0[f]:f32[s:4,a:4]}
```

5.7. Enum Signatures

Enums include their underlying type:

```
enum class Color : uint8_t { Red, Green, Blue };
// enum[s:1,a:1]<u8[s:1,a:1]>
```

5.8. Array Signatures

Arrays include element type and count:

```
int32_t arr[4];
// array[s:16,a:4]<i32[s:4,a:4],4>
```

Special case for `char` arrays:

```
char buf[64];
// bytes[s:64,a:1]
```

5.9. Bitfield Signatures

Bitfields use a special format:

```
struct Flags {
    uint8_t a : 3;
    uint8_t b : 5;
};
// struct[s:1,a:1]{@0.0[a]:bits<3,u8[s:1,a:1]>,@0.3[b]:bits<5,u8[s:1,a:1]>}
```

- `@0.0` - byte 0, bit 0
- `@0.3` - byte 0, bit 3
- `bits<WIDTH,TYPE>` - bitfield width and underlying type

5.10. Anonymous Members

Anonymous unions/structs get placeholder names:

```
struct Mixed {
    int32_t x;
    union { int32_t a; float b; };  // Anonymous union
};
// struct[s:8,a:4]{@0[x]:i32[s:4,a:4],@4[<anon:0>]:union[s:4,a:4]{...}}
```

5.11. Signature Comparison Rules

Two signatures are equal if and only if:

1. Platform prefix matches
2. Type category matches
3. Size and alignment match
4. All field offsets match
5. All field names match
6. All field types match (recursively)

Field **names** are part of the signature. Different names = different signature.

Chapter 6. Chapter 5: Real-World Applications

Practical use cases for TypeLayout.

6.1. Application 1: Shared Memory IPC

Multiple processes share data through memory-mapped regions.

6.1.1. The Problem

```
// Process A writes
SharedData* data = map_shared_memory("/my_shm");
data->value = 42;

// Process B reads
SharedData* data = map_shared_memory("/my_shm");
int v = data->value;  // Is this correct?
```

If Process A and B were compiled differently, `value` might be at different offsets.

6.1.2. The Solution

```
#include <boost/typelayout.hpp>

struct SharedData {
    uint32_t magic;
    uint64_t layout_hash;  // Layout verification
    // ... actual data ...
    int32_t  value;
    double   temperature;
};

// At compile time, lock down the expected layout
constexpr auto EXPECTED_HASH = get_layout_hash<SharedData>();

void* map_shared_memory(const char* name) {
    void* ptr = /* mmap or shm_open */;
    auto* header = static_cast<SharedData*>(ptr);

    // Runtime verification
    if (header->layout_hash != EXPECTED_HASH) {
        throw std::runtime_error("Layout mismatch in shared memory!");
    }
    return ptr;
}
```

```
void init_shared_memory(SharedData* data) {
    data->magic = 0x12345678;
    data->layout_hash = EXPECTED_HASH;
}
```

6.2. Application 2: File Format Versioning

Game save files must remain compatible across versions.

6.2.1. The Problem

```
// Version 1.0
struct SaveFile_v1 {
    uint32_t version;
    char player_name[32];
    int32_t score;
};

// Version 2.0 - added new field
struct SaveFile_v2 {
    uint32_t version;
    char player_name[32];
    int32_t score;
    uint64_t playtime;  // NEW!
};
```

How do you handle old save files?

6.2.2. The Solution

```
// Define known layouts with their signatures
constexpr auto V1_SIGNATURE =
    "[64-le]struct[s:40,a:4]{@0[version]:u32[s:4,a:4],"
    "@4[player_name]:bytes[s:32,a:1],@36[score]:i32[s:4,a:4]}";

constexpr auto V2_SIGNATURE =
    "[64-le]struct[s:48,a:8]{@0[version]:u32[s:4,a:4],"
    "@4[player_name]:bytes[s:32,a:1],@36[score]:i32[s:4,a:4],"
    "@40[playtime]:u64[s:8,a:8]}";

// Compile-time verification
static_assert(LayoutMatch<SaveFile_v1, V1_SIGNATURE>);
static_assert(LayoutMatch<SaveFile_v2, V2_SIGNATURE>);

// Runtime loader
SaveData load_save(const char* path) {
    FileHeader header = read_header(path);

    switch (header.version) {
        case 1: return load_v1(path);
        case 2: return load_v2(path);
        default: throw std::runtime_error("Unknown save version");
    }
}
```

6.3. Application 3: Network Protocol

Client and server must agree on message layouts.

6.3.1. Protocol Definition

```
// protocol.hpp - shared between client and server
namespace protocol {

struct MessageHeader {
    uint32_t magic;       // 0xDEADBEEF
    uint16_t type;        // Message type
    uint16_t flags;       // Flags
    uint32_t length;      // Payload length
    uint64_t timestamp;   // Unix timestamp
};

struct LoginRequest {
    MessageHeader header;
    char username[32];
    char password_hash[64];
};

struct LoginResponse {
    MessageHeader header;
    uint32_t result_code;
    uint64_t session_id;
};

// Compile-time layout contracts
static_assert(get_layout_signature<MessageHeader>() ==
    "[64-le]struct[s:24,a:8]{"
    "@0[magic]:u32[s:4,a:4],"
    "@4[type]:u16[s:2,a:2],"
    "@6[flags]:u16[s:2,a:2],"
    "@8[length]:u32[s:4,a:4],"
    "@16[timestamp]:u64[s:8,a:8]}");

} // namespace protocol
```

6.3.2. Version Negotiation

```cpp
// Include layout hash in handshake
struct Handshake {
    uint32_t protocol_version;
    uint64_t header_layout_hash;
};

void connect(Socket& socket) {
    Handshake hs;
    hs.protocol_version = 1;
    hs.header_layout_hash = get_layout_hash<MessageHeader>();

    socket.send(&hs, sizeof(hs));

    Handshake server_hs;
    socket.recv(&server_hs, sizeof(server_hs));

    if (server_hs.header_layout_hash != hs.header_layout_hash) {
        throw std::runtime_error("Protocol layout mismatch!");
    }
}
```

6.4. Application 4: Hardware Register Mapping

Embedded systems access hardware through memory-mapped registers.

6.4.1. Register Definition

```cpp
// SPI controller registers (hypothetical)
struct SPIRegisters {
    volatile uint32_t control;    // @0x00: Control register
    volatile uint32_t status;     // @0x04: Status register
    volatile uint32_t data;       // @0x08: Data register
    volatile uint32_t clock_div;  // @0x0C: Clock divider
};

// Verify against hardware specification
static_assert(sizeof(SPIRegisters) == 16);
static_assert(get_layout_signature<SPIRegisters>() ==
    "[32-le]struct[s:16,a:4]{"
    "@0[control]:u32[s:4,a:4],"
    "@4[status]:u32[s:4,a:4],"
    "@8[data]:u32[s:4,a:4],"
    "@12[clock_div]:u32[s:4,a:4]}");

SPIRegisters* spi = reinterpret_cast<SPIRegisters*>(0x40001000);
```

6.5. Application 5: ABI Stability

Library maintainers need to ensure ABI stability.

6.5.1. Public API Types

```
// public_api.hpp

// ABI-stable types - changes break compatibility
struct [[nodiscard]] Result {
    int32_t  error_code;
    uint64_t data;
};

struct Config {
    uint32_t flags;
    uint32_t timeout_ms;
    char     name[64];
};

// Document and enforce ABI
namespace abi {
    // Current ABI version
    constexpr int VERSION = 1;

    // Layout hashes for verification
    constexpr uint64_t RESULT_HASH = get_layout_hash<Result>();
    constexpr uint64_t CONFIG_HASH = get_layout_hash<Config>();

    // CI can verify these don't change unexpectedly
    static_assert(RESULT_HASH == 0x1234567890ABCDEFull,
        "ABI break: Result layout changed!");
}
```

6.6. Summary

TypeLayout is useful whenever you need to:

Scenario	Key Technique
Shared Memory	Hash in header + runtime check
File Formats	Version-specific signatures
Network Protocols	Layout hash in handshake
Hardware Registers	Static assert against spec
ABI Stability	CI-verified hash constants

Chapter 7. Chapter 6: Beyond the Basics

Advanced features: inheritance, polymorphism, bitfields, and concepts.

7.1. Inheritance

TypeLayout fully supports class inheritance, including private members.

7.1.1. Single Inheritance

```
class Entity {
public:
    Entity(uint64_t id) : id_(id) {}
    uint64_t getId() const { return id_; }
private:
    uint64_t id_;
};

class Player : public Entity {
public:
    Player(uint64_t id, int32_t score) : Entity(id), score_(score) {}
    int32_t getScore() const { return score_; }
private:
    int32_t score_;
};

// Signature includes base class
constexpr auto sig = get_layout_signature<Player>();
//
class[s:16,a:8,inherited]{@0[base]:struct[s:8,a:8]{@0[id_]:u64[s:8,a:8]},@8[score_]:i3
2[s:4,a:4]}
```

The `inherited` marker indicates base class presence.

7.1.2. Multiple Inheritance

```
struct Movable { float x, y; };
struct Renderable { uint32_t sprite_id; };

struct GameObject : Movable, Renderable {
    uint32_t object_id;
};

constexpr auto sig = get_layout_signature<GameObject>();
// Includes both base classes with their offsets
```

7.1.3. Virtual Inheritance

```
struct VirtualBase { int32_t value; };
struct Left : virtual VirtualBase { int32_t left_data; };
struct Right : virtual VirtualBase { int32_t right_data; };
struct Diamond : Left, Right { int32_t diamond_data; };

// Virtual bases are marked with [vbase]
// @N[vbase]:struct{...}
```

7.2. Polymorphism

Polymorphic classes (with virtual functions) are handled correctly.

```
class IShape {
public:
    virtual ~IShape() = default;
    virtual double area() const = 0;
};

class Circle : public IShape {
public:
    Circle(double r) : radius_(r) {}
    double area() const override { return 3.14159 * radius_ * radius_; }
private:
    double radius_;
};

constexpr auto sig = get_layout_signature<Circle>();
// class[s:16,a:8,polymorphic,inherited]{...}
```

The `polymorphic` marker indicates:

- A vtable pointer exists (typically at offset 0)
- The class has virtual functions
- Size includes vtable pointer (8 bytes on 64-bit)

7.3. Bitfields

Bitfields are represented with bit-level precision.

```
struct PacketFlags {
    uint8_t version : 4;    // Bits 0-3
    uint8_t type : 3;       // Bits 4-6
    uint8_t urgent : 1;     // Bit 7
    uint8_t priority : 4;   // Bits 8-11 (second byte)
    uint8_t reserved : 4;   // Bits 12-15
};

constexpr auto sig = get_layout_signature<PacketFlags>();
// struct[s:2,a:1]{
//   @0.0[version]:bits<4,u8[s:1,a:1]>,
//   @0.4[type]:bits<3,u8[s:1,a:1]>,
//   @0.7[urgent]:bits<1,u8[s:1,a:1]>,
//   @1.0[priority]:bits<4,u8[s:1,a:1]>,
//   @1.4[reserved]:bits<4,u8[s:1,a:1]>
// }
```

Bitfield format: `@BYTE.BIT[name]:bits<WIDTH,UNDERLYING>`

7.4. Anonymous Members

Anonymous unions and structs get placeholder names.

```
struct Variant {
    uint32_t type;
    union {
        int32_t  as_int;
        float    as_float;
        void*    as_ptr;
    };  // Anonymous union
};

constexpr auto sig = get_layout_signature<Variant>();
// struct[s:16,a:8]{@0[type]:u32[s:4,a:4],@8[<anon:0>]:union[s:8,a:8]{...}}
```

7.5. Concepts Integration

TypeLayout provides C++20 concepts for type constraints.

7.5.1. LayoutSupported

Check if a type can have its layout computed:

```
template<typename T>
    requires LayoutSupported<T>
void serialize(const T& value, Buffer& buf) {
    buf.write(&value, sizeof(T));
}

// Works with: structs, classes, unions, enums, primitives, arrays
// Fails with: void, function types, incomplete types
```

7.5.2. LayoutCompatible

Check if two types have identical layouts:

```
template<typename T, typename U>
    requires LayoutCompatible<T, U>
void safe_copy(const T& src, U& dst) {
    std::memcpy(&dst, &src, sizeof(T));
}

// Only compiles if T and U have exactly the same layout signature
```

7.5.3. LayoutMatch

Check if a type matches a specific signature:

```
template<typename T>
    requires LayoutMatch<T, "[64-
le]struct[s:8,a:4]{@0[x]:i32[s:4,a:4],@4[y]:i32[s:4,a:4]}">
void process_point(const T& point) {
    // Guaranteed layout
}
```

7.5.4. LayoutHashMatch

Check if a type matches a specific hash:

```
template<typename T>
    requires LayoutHashMatch<T, 0xABCD1234DEADBEEF>
void load_from_file(T& data, const char* path) {
    // Layout verified at compile time
}
```

7.6. Practical Example: Type-Safe Memory Pool

```
template<typename T>
    requires LayoutSupported<T>
class MemoryPool {
public:
    // Store layout hash for verification
    static constexpr uint64_t LAYOUT_HASH = get_layout_hash<T>();

    T* allocate() {
        // ...
    }
```

```cpp
    void deallocate(T* ptr) {
        // ...
    }

    // Verify at runtime (e.g., when loading serialized pool)
    bool verify_layout(uint64_t stored_hash) const {
        return stored_hash == LAYOUT_HASH;
    }
};
```

7.7. Private Members

TypeLayout uses P2996's `access_context::unchecked()` to reflect private members.

```cpp
class SecretData {
private:
    uint64_t secret_key_;
    int32_t  secret_value_;
};

// Works! Private members are fully visible
constexpr auto sig = get_layout_signature<SecretData>();
// struct[s:16,a:8]{@0[secret_key_]:u64[s:8,a:8],@8[secret_value_]:i32[s:4,a:4]}
```

No `friend` declarations needed. No workarounds. Just works.

Chapter 8. Chapter 7: Under the Hood

How TypeLayout uses C++26 static reflection (P2996) internally.

8.1. Introduction to P2996

P2996 "Reflection for C++26" introduces compile-time type introspection via:

- `^T` - The "reflection operator" that creates a reflection of type T
- `std::meta::info` - Opaque handle representing reflected entities
- `[: refl :]` - The "splice" operator that converts reflection back to code

```
#include <experimental/meta>

struct Point { int x, y; };

consteval void example() {
    // Get reflection of Point
    constexpr auto refl = ^Point;

    // Query its properties
    static_assert(std::meta::is_class_type(refl));

    // Get members
    auto members = std::meta::nonstatic_data_members_of(refl);
}
```

8.2. Key P2996 APIs Used

TypeLayout relies on these core P2996 functions:

8.2.1. `nonstatic_data_members_of`

Returns a list of non-static data members:

```
template<typename T>
consteval std::size_t get_member_count() {
    using namespace std::meta;
    auto members = nonstatic_data_members_of(^T, access_context::unchecked());
    return members.size();
}
```

8.2.2. `offset_of`

Gets the byte (and bit) offset of a member:

```
template<typename T, size_t Index>
consteval size_t get_field_offset() {
    using namespace std::meta;
    auto members = nonstatic_data_members_of(^T, access_context::unchecked());
    return offset_of(members[Index]).bytes;
}
```

8.2.3. `type_of`

Gets the type of a reflected entity:

```
template<typename T, size_t Index>
using FieldType = [: type_of(
    nonstatic_data_members_of(^T, access_context::unchecked())[Index]
) :];
```

8.2.4. `identifier_of`

Gets the name of a reflected entity:

```
consteval std::string_view get_field_name(std::meta::info member) {
    if (std::meta::has_identifier(member)) {
        return std::meta::identifier_of(member);
    }
    return "<anonymous>";
}
```

8.2.5. `bases_of`

Gets base classes of a type:

```
template<typename T>
consteval bool has_bases() {
    using namespace std::meta;
    return bases_of(^T, access_context::unchecked()).size() > 0;
}
```

8.3. Access Control: The Magic of `unchecked()`

Traditional reflection (like `offsetof`) fails on private members:

```
class Secret {
private:
    int value_;
};

// Error: 'value_' is private
// offsetof(Secret, value_);
```

P2996 solves this with `access_context::unchecked()`:

```
auto members = nonstatic_data_members_of(^Secret,
    access_context::unchecked());  // Bypasses access control!

auto offset = offset_of(members[0]).bytes;  // Works!
```

This is by design - reflection for introspection should see everything.

8.4. How TypeLayout Generates Signatures

8.4.1. Step 1: Iterate Members

```
template<typename T>
consteval auto get_fields_signature() {
    constexpr auto members = nonstatic_data_members_of(
        ^T, access_context::unchecked());

    // Process each member...
    return concatenate_signatures<T>(
        std::make_index_sequence<members.size()>{}
    );
}
```

8.4.2. Step 2: Build Field Signature

```
template<typename T, size_t Index>
consteval auto get_field_signature() {
    constexpr auto member = nonstatic_data_members_of(
        ^T, access_context::unchecked())[Index];

    using FieldType = [: type_of(member) :];
    constexpr size_t offset = offset_of(member).bytes;
    constexpr auto name = identifier_of(member);

    // Build: @OFFSET[NAME]:TYPE_SIGNATURE
    return CompileString{"@"} +
           to_string(offset) +
           CompileString{"["} +
           CompileString{name} +
           CompileString{"]:"} +
            TypeSignature<FieldType>::calculate();
}
```

8.4.3. Step 3: Handle Special Cases

Bitfields:

```
if constexpr (is_bit_field(member)) {
    auto bit_off = offset_of(member);
    // Use bit_off.bytes and bit_off.bits
    // Format: @BYTE.BIT[name]:bits<WIDTH,TYPE>
}
```

Base classes:

```
template<typename T>
consteval auto get_bases_signature() {
    auto bases = bases_of(^T, access_context::unchecked());
    // For each base, get offset and signature
}
```

8.5. The CompileString Type

TypeLayout uses a custom compile-time string type:

```
template<size_t N>
struct CompileString {
    char value[N];
    static constexpr size_t size = N;

    consteval CompileString(const char (&str)[N]) {
        std::copy_n(str, N, value);
    }

    // Concatenation
    template<size_t M>
    consteval auto operator+(const CompileString<M>& other) const {
        CompileString<N + M - 1> result;
        // Copy this, then other...
        return result;
    }
};
```

This enables building signatures entirely at compile time.

8.6. Recursion and Nested Types

For nested structs, TypeLayout recursively generates signatures:

```
struct Inner { int32_t value; };
struct Outer { Inner inner; int32_t extra; };

// Outer's signature includes Inner's full signature:
// struct[s:8,a:4]{@0[inner]:struct[s:4,a:4]{@0[value]:i32},@4[extra]:i32}
```

This is achieved by `TypeSignature<FieldType>::calculate()` recursing into nested types.

8.7. Performance Considerations

All computation happens at compile time:

- No runtime overhead
- No binary size increase (signatures are constexpr values)
- Build time: proportional to type complexity

Typical impact:

- Simple structs: ~0ms
- Complex hierarchies: ~10-50ms
- Very deep nesting: may hit compiler limits

8.8. Compiler Support

Currently requires Bloomberg's experimental Clang with P2996:

```
# Build with P2996 toolchain
clang++ -std=c++26 -freflection my_code.cpp
```

As P2996 gets standardized, expect support in:

- GCC 15+ (planned)
- Clang 19+ (planned)
- MSVC (TBD)

8.9. Summary

TypeLayout leverages P2996 to:

1. **Reflect** all members (including private) at compile time

2. **Extract** offsets, names, and types automatically
3. **Generate** human-readable signatures with zero runtime cost
4. **Verify** layouts at compile time via `static_assert` and concepts

The result: automatic, complete, and efficient layout introspection.

Chapter 9. Appendix A: Quick Reference

9.1. Signature Format Cheat Sheet

Format	Meaning
`[64-le]`	64-bit little-endian platform
`struct[s:N,a:M]`	Struct with size N, alignment M
`class[s:N,a:M,polymorphic]`	Class with vtable
`class[s:N,a:M,inherited]`	Class with base classes
`@N[name]:TYPE`	Field at offset N
`@N.B[name]:bits<W,T>`	Bitfield at byte N, bit B, width W
`i32[s:4,a:4]`	Signed 32-bit integer
`u64[s:8,a:8]`	Unsigned 64-bit integer
`f64[s:8,a:8]`	Double precision float
`ptr[s:8,a:8]`	Pointer
`array[s:N,a:M]<T,C>`	Array of C elements of type T
`bytes[s:N,a:1]`	Byte array (char[N])
`enum[s:N,a:M]<T>`	Enum with underlying type T

9.2. API Quick Reference

```
// Core functions
get_layout_signature<T>()     // Get full signature
get_layout_hash<T>()          // Get 64-bit hash

// Concepts
LayoutSupported<T>            // Can compute layout?
LayoutCompatible<T, U>        // Same layout?
LayoutMatch<T, "sig">         // Matches signature?
LayoutHashMatch<T, hash>      // Matches hash?

// Macro
BOOST_TYPELAYOUT_ASSERT(T, "sig")  // Assert at compile time
```

Chapter 10. Appendix B: Further Reading

10.1. C++ Standards Papers

- **P2996** - Reflection for C++26
- **P2320** - The Syntax of Static Reflection
- **P1240** - Scalable Reflection

10.2. Tools & Resources

- Bloomberg’s P2996 Clang Fork: https://github.com/bloomberg/clang-p2996
- cppreference.com - Memory layout documentation
- pahole - Poke-A-Hole, data structure analysis tool

10.3. Related Projects

- Boost.PFR - Simple struct reflection (C+14)
- Magic Enum - Enum reflection
- refl-cpp - Header-only reflection library

Chapter 11. Appendix C: Troubleshooting

11.1. Common Issues

11.1.1. "Compiler does not support P2996"

```
error: unknown reflection operator '^'
```

Solution: Ensure you're using Bloomberg's P2996 Clang fork with `-std=c++26 -freflection` flags.

11.1.2. "Layout hash mismatch at runtime"

Possible causes:

1. Different compilers were used for producer and consumer
2. Different compiler flags (e.g., `-fpack-struct`)
3. Different platforms (32-bit vs 64-bit)

Solution: Ensure all components are built with identical toolchains and flags.

11.1.3. "Signature too long for static_assert"

For complex types, signatures can exceed compiler limits.

Solution: Use hash comparison instead:

```
static_assert(get_layout_hash<ComplexType>() == 0x1234567890ABCDEF);
```

11.2. Getting Help

- GitHub Issues: https://github.com/boost/typelayout/issues
- Boost Mailing List: boost-users@lists.boost.org
- Stack Overflow: Tag `[boost-typelayout]`

Chapter 12. Glossary

ABI

Application Binary Interface. The low-level interface between compiled code modules.

Alignment

The byte boundary on which a type must be placed in memory.

Bitfield

A struct member with a specified bit width.

Endianness

The byte order used to store multi-byte values (little-endian or big-endian).

Layout Signature

A string representation of a type’s memory layout.

Padding

Unused bytes inserted by the compiler to satisfy alignment requirements.

P2996

The C proposal for static reflection, targeting C26.

Splice

The operation that converts a reflection back into code (`[: expr :]`).

Standard-layout

A C++ type category with guaranteed memory layout properties.

vtable

Virtual function table, used for runtime polymorphism.

TypeLayout: Type Layout Verification for C++26

Version 1.0 | February 2026